Advanced Data Structures and Algorithm Analysis

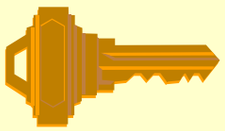
丁尧相
浙江大学

Fall & Winter 2025
Lecture 6

Dynamic Programming

- Shortest path in DAGs
- Maximum subsequence sum
- Optimal binary search trees
- 0-1 knapsack
- Matrix multiplication
- Product assembly
- Shortest path, revisited

Solve sub-problems just once and save answers in a table



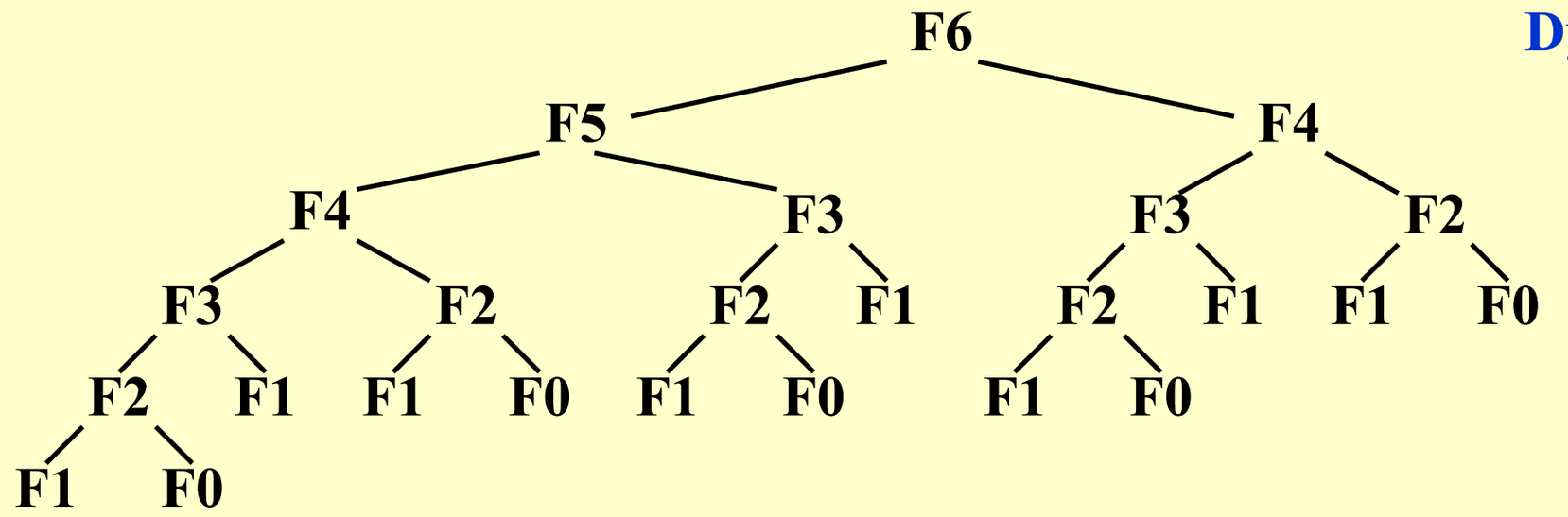
Use a table instead of recursion

1. **Fibonacci Numbers:** $F(N) = F(N-1) + F(N-2)$

```
int  Fib( int N )
{
    if ( N <= 1 )
        return  1;
    else
        return  Fib( N - 1 ) + Fib( N - 2 );
}
```

$$T(N) \geq T(N-1) + T(N-2)$$

$$\rightarrow \quad T(N) \geq F(N)$$



Trouble-maker: **The growth of redundant calculations is explosive.**

Solution: **Record the two most recently computed values to avoid recursive calls.**

```
int  Fibonacci ( int N )
{   int  i, Last, NextToLast, Answer;
    if ( N <= 1 )  return  1;
    Last = NextToLast = 1;    /* F(0) = F(1) = 1 */
    for ( i = 2; i <= N; i++ ) {
        Answer = Last + NextToLast;   /* F(i) = F(i-1) + F(i-2) */
        NextToLast = Last; Last = Answer;  /* update F(i-1) and F(i-2) */
    }  /* end-for */
    return  Answer;
}
```

$$T(N) = \mathbf{O}(N)$$

Those who cannot remember the past are condemned to repeat it.

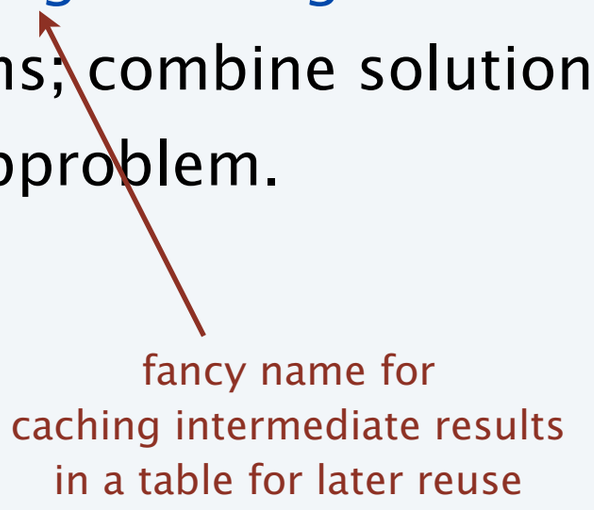
- Dynamic Programming

Algorithmic paradigms

Greed. Process the input in some order, myopically making irrevocable decisions.

Divide-and-conquer. Break up a problem into independent subproblems; solve each subproblem; combine solutions to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form solution to large subproblem.



Dynamic programming history

Bellman. Pioneered the systematic study of dynamic programming in 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense had pathological fear of mathematical research.
- Bellman sought a "dynamic" adjective to avoid conflict.

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

Programming?

The origin of the term *dynamic programming* has very little to do with writing code. It was first coined by Richard Bellman in the 1950s, a time when computer programming was an esoteric activity practiced by so few people as to not even merit a name. Back then programming meant "planning," and "dynamic programming" was conceived to optimally plan multistage processes. The dag of Figure 6.2 can be thought of as describing the possible ways in which such a process can evolve: each node denotes a state, the leftmost node is the starting point, and the edges leaving a state represent possible actions, leading to different states in the next unit of time.

The etymology of *linear programming*, the subject of Chapter 7, is similar.

Dynamic programming applications

Application areas.

- Computer science: AI, compilers, systems, graphics, theory, ...
- Operations research.
- Information theory.
- Control theory.
- Bioinformatics.

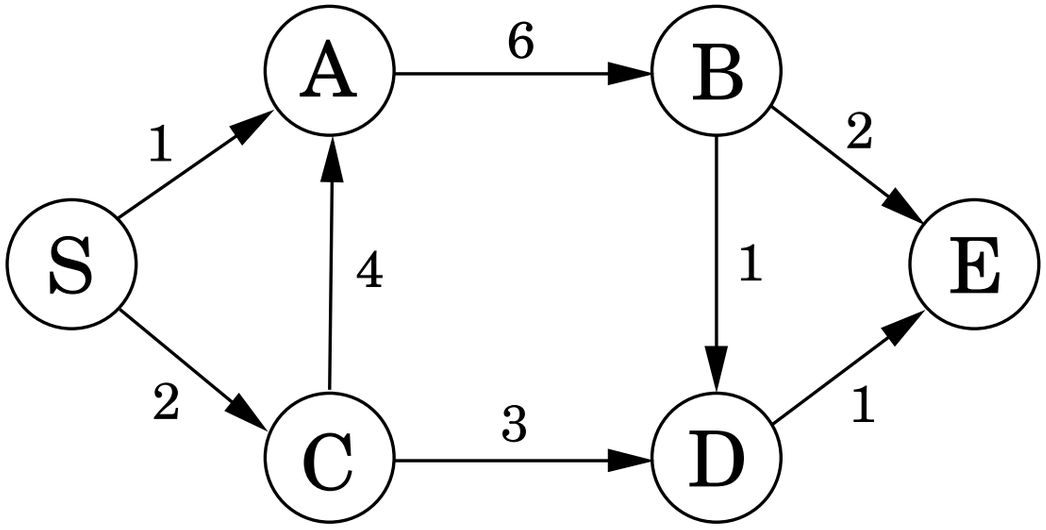
Some famous dynamic programming algorithms.

- Avidan–Shamir for seam carving.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Bellman–Ford–Moore for shortest path.
- Knuth–Plass for word wrapping text in TEX.
- Cocke–Kasami–Younger for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman for sequence alignment.

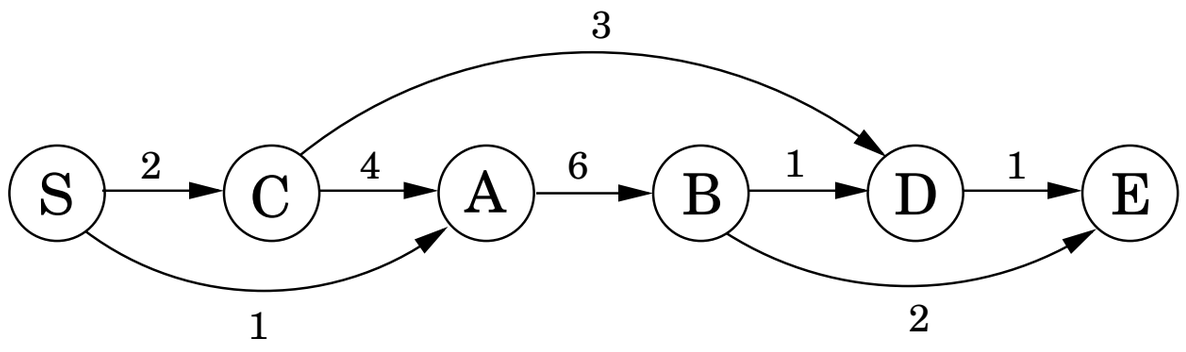
Dynamic Programming

- Shortest path in DAGs
- Longest increasing subsequences
- Optimal binary search trees
- 0-1 knapsack
- Matrix multiplication
- Product assembly
- Shortest path, revisited

Shortest Path in DAGs

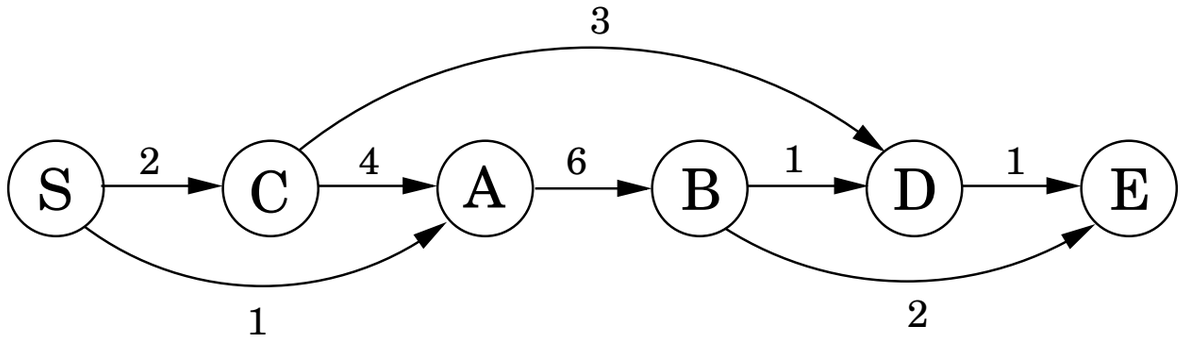


Calculate the shortest path from S to E.
Why this is an easy task?



The problem can be linearized to form this transition graph.

Shortest Path in DAGs



Solving the global problem can be realized by solving previous sub-problems.
Take node D as an example.

$$\texttt{dist}(D) = \min\{\texttt{dist}(B) + 1, \texttt{dist}(C) + 3\}.$$

initialize all $\texttt{dist}(\cdot)$ values to ∞
$\texttt{dist}(s) = 0$
for each $v \in V \backslash \{s\}$, in linearized order:
    $\texttt{dist}(v) = \min_{(u,v)\in E}\{\texttt{dist}(u) + l(u, v)\}$

Remember the solutions of sub-problems will reduce the cost of larger global problems.
The DAG structure is implicit but common in DP problems.
Note: DP does not necessarily assume DAG structure.
While we always assume this in our course.

Dynamic Programming

- Shortest path in DAGs
- Maximum subsequence sum
- Optimal binary search trees
- 0-1 knapsack
- Matrix multiplication
- Product assembly
- Shortest path, revisited

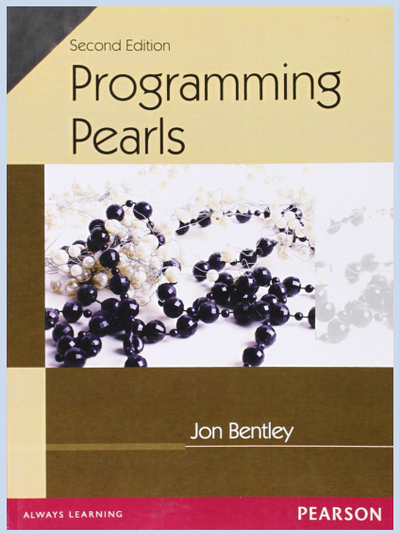
Maximum subarray problem

Goal. Given an array x of n integer (positive or negative), find a contiguous subarray whose sum is maximum.

12	5	−1	31	−61	59	26	−53	58	97	−93	−23	84	−15	6

187

Applications. Computer vision, data mining, genomic sequence analysis, technical job interviews, ...



Maximum subarray problem

Goal. Given an array x of n integer (positive or negative), find a contiguous subarray whose sum is maximum.

					i				j					
12	5	−1	31	−61	59	26	−53	58	97	−93	−23	84	−15	6

187

The divide-and-conquer solution takes $O(n \log n)$ time.

Brute-force algorithm.

- For each i and j : computer $a[i] + a[i+1] + \ldots + a[j]$.
- Takes $\Theta(n^3)$ time.

Use the idea of remembering the past.

KADANE'S ALGORITHM

Def. $OPT(i)$ = max sum of any subarray of x whose rightmost index is i.

Goal. $\max_i \; OPT(i)$

Bellman equation.

$$OPT(i) \;=\; \begin{cases} x_1 & \text{if } i = 1 \\ \max\{\, x_i,\; x_i + OPT(i-1) \,\} & \text{if } i > 1 \end{cases}$$

take only element i

take element i together with best subarray ending at index $i-1$

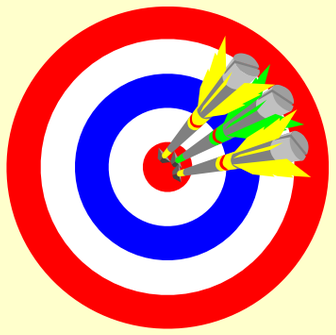
Running time. $O(n)$.

Dynamic Programming

- Shortest path in DAGs
- Longest increasing subsequences
- Optimal binary search trees
- 0-1 knapsack
- Matrix multiplication
- Product assembly
- Shortest path, revisited

3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)

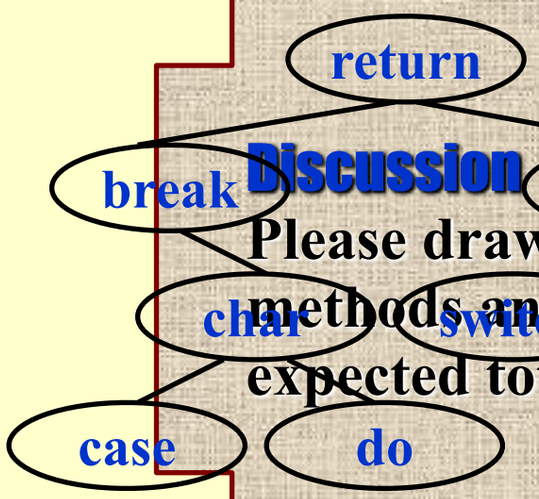


Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

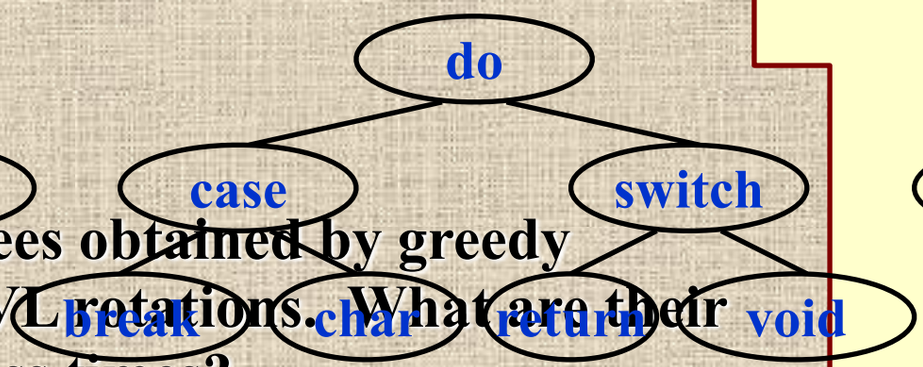
word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

Greedy Method



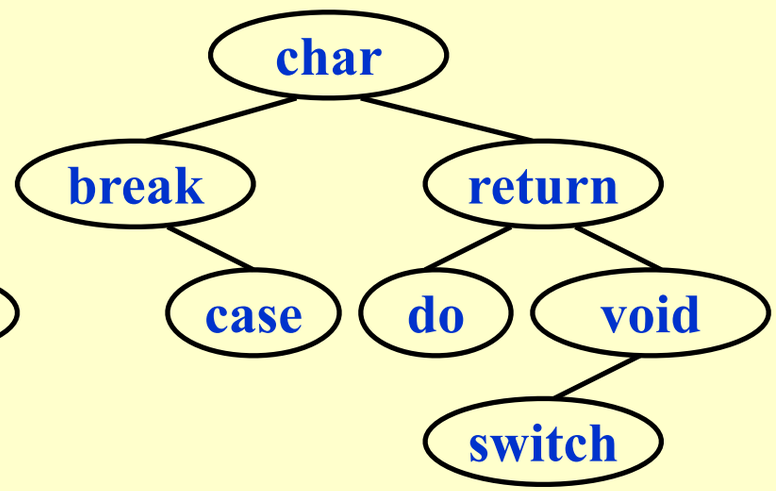
Time = 2.43

AVL Method



Time = 2.70

Optimal Tree



Time = 2.15

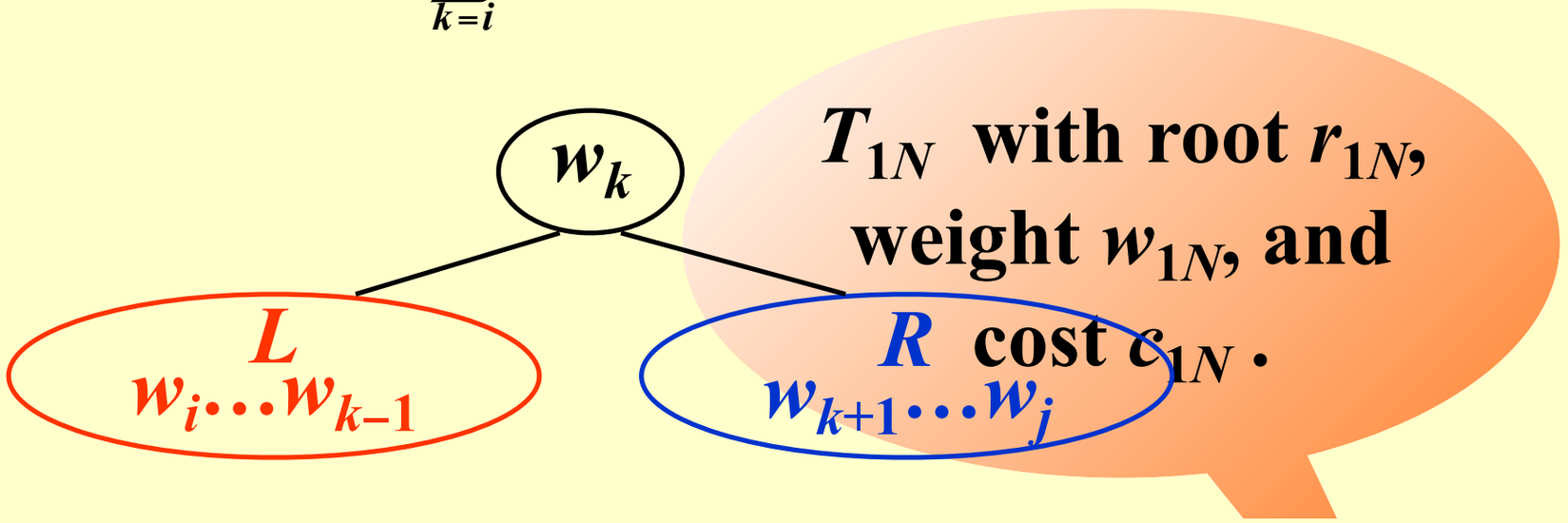
Discussion 10:
Please draw the trees obtained by greedy methods and by AVL rotations. What are their expected total access times?

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} $(c_{ii} = 0$ if $j < i)$

r_{ij} ::= root of T_{ij}

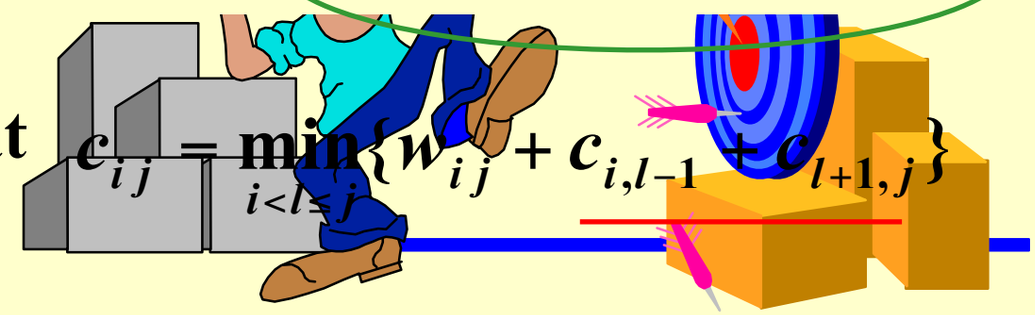
w_{ij} ::= weight of $T_{ij} = \sum_{k=i}^{j} p_k$ $(w_{ii} = p_i)$



$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R)$

$= p_k + c_{i,k-1} + c_{k+1,j} + w_{i,k-1} + w_{k+1,j} = w_{ij} + c_{i,k-1} + c_{k+1,j}$

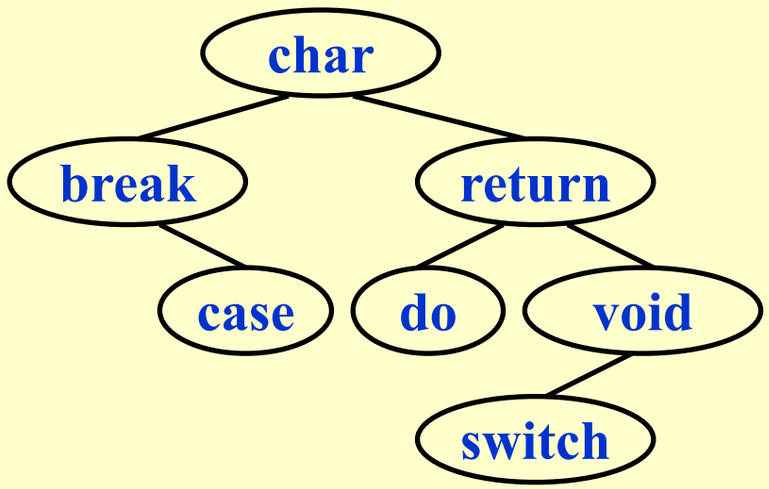
T_{ij} is optimal $\Rightarrow$ $r_{ij} = k$ is such that $c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$T(N) = O(N^3)$$

Please read Weiss 10.33 on p.419 for an O(N^2) algorithm.

Dynamic Programming

- Shortest path in DAGs
- Longest increasing subsequences
- Optimal binary search trees
- 0-1 knapsack
- Matrix multiplication
- Product assembly
- Shortest path, revisited

Knapsack problem

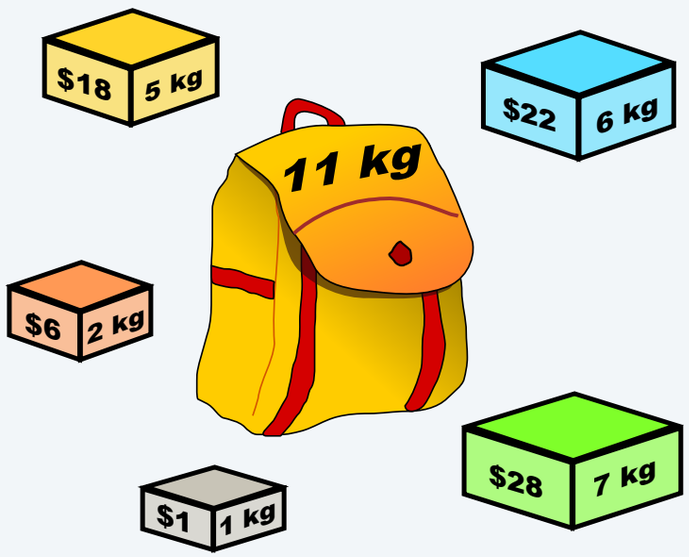
Goal. Pack knapsack so as to maximize total value of items taken.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of W.

Ex. The subset $\{ 1, 2, 5 \}$ has value \$35 (and weight 10).

Ex. The subset $\{ 3, 4 \}$ has value \$40 (and weight 11).

Assumption. All values and weights are integral.



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance (weight limit W = 11)

weights and values can be arbitrary positive integers

Dynamic programming: quiz 2

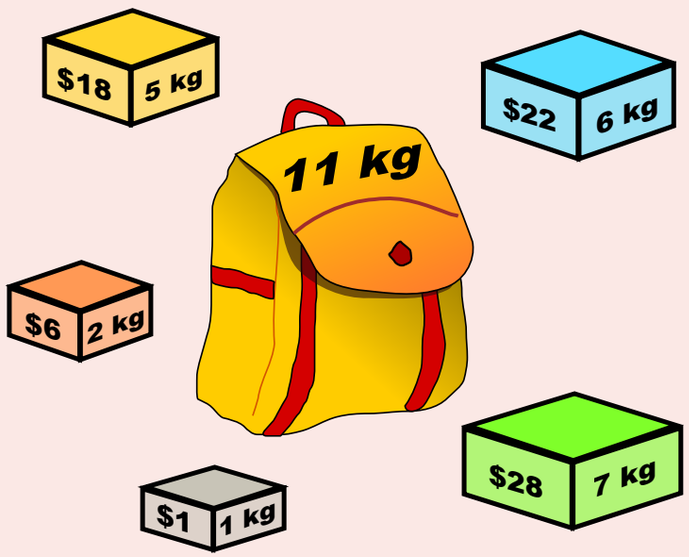
Which algorithm solves knapsack problem?

A. Greedy-by-value: repeatedly add item with maximum v_i.

B. Greedy-by-weight: repeatedly add item with minimum w_i.

C. Greedy-by-ratio: repeatedly add item with maximum ratio v_i / w_i.

D. None of the above.



i	v_i	w_i
1	$1	1 kg
2	$6	2 kg
3	$18	5 kg
4	$22	6 kg
5	$28	7 kg

knapsack instance
(weight limit W = 11)

Dynamic programming: quiz 3

Which subproblems?

A. $OPT(w)$ = optimal value of knapsack problem with weight limit w.

B. $OPT(i)$ = optimal value of knapsack problem with items $1, \ldots, i$.

C. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$ subject to weight limit w.

D. Any of the above.

Dynamic programming: two variables

Def. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, subject to weight limit w.

Goal. $OPT(n, W)$.

Case 1. $OPT(i, w)$ does not select item i.

possibly because $w_i > w$

- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i-1 \}$ subject to weight limit w.

Case 2. $OPT(i, w)$ selects item i.

optimal substructure property
(proof via exchange argument)

- Collect value v_i.
- New weight limit $= w - w_i$.
- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i-1 \}$ subject to new weight limit.

Bellman equation.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w),\ v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$$

Knapsack problem: bottom-up dynamic programming demo

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1,w-w_i\} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, …, i

	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = optimal value of knapsack problem with items 1, …, i, subject to weight limit w

Knapsack problem: bottom-up dynamic programming

KNAPSACK$(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO W

  $M[0, w] \leftarrow 0$.

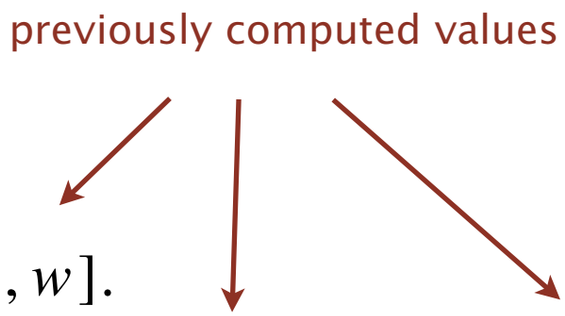
FOR $i = 1$ TO n

  FOR $w = 0$ TO W

    IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

    ELSE $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w - w_i] \}$.

RETURN $M[n, W]$.



$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{\, OPT(i-1, w),\ \ v_i + OPT(i-1, w - w_i)\,\} & \text{otherwise} \end{cases}$$

Knapsack problem: running time

Theorem. The DP algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(n\,W)$ time and $\Theta(n\,W)$ space.

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\,W)$ table entries.
- After computing optimal values, can trace back to find solution: $OPT(i, w)$ takes item i iff $M[i, w] > M[i-1, w]$. ▪

weights are integers between 1 and W

Remarks.

- Algorithm depends critically on assumption that weights are integral.
- Assumption that values are integral was not used.

Dynamic Programming

Instance: n items i=1,2, ..., n;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

define:

$A(i, v)$ = minimum total weight of $S \subseteq \{1,2, \ldots, i\}$
with total value *exactly* v

$A(i, v) = \infty$ if no such S exists

Dynamic Programming

Instance: n items i=1,2, ..., n;
weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;
knapsack capacity $B \in \mathbb{Z}^+$;

$A(i, v)$ = minimum total weight of $S \subseteq \{1,2, \ldots, i\}$
with total value *exactly* v

recursion:

$$A(i, v) = \min\{ A(i\text{-}1, v),\ A(i\text{-}1, v\text{-}v_i) + w_i \} \quad \text{for } i > 1$$

$$A(1, v) = \begin{cases} w_1 & \text{if } v = v_1 \\ \infty & \text{otherwise} \end{cases}$$

$1 \le i \le n, \quad 1 \le v \le V = \sum_i v_i$

Dynamic programming:
table size $O(nV)$
time complexity $O(nV)$

Dynamic programming: quiz 4

Does there exist a poly-time algorithm for the knapsack problem?

A. Yes, because the DP algorithm takes $\Theta(n\,W)$ time.

B. No, because $\Theta(n\,W)$ is not a polynomial function of the input size.

C. No, because the problem is **NP**-hard.

D. Unknown.

Knapsack Problem

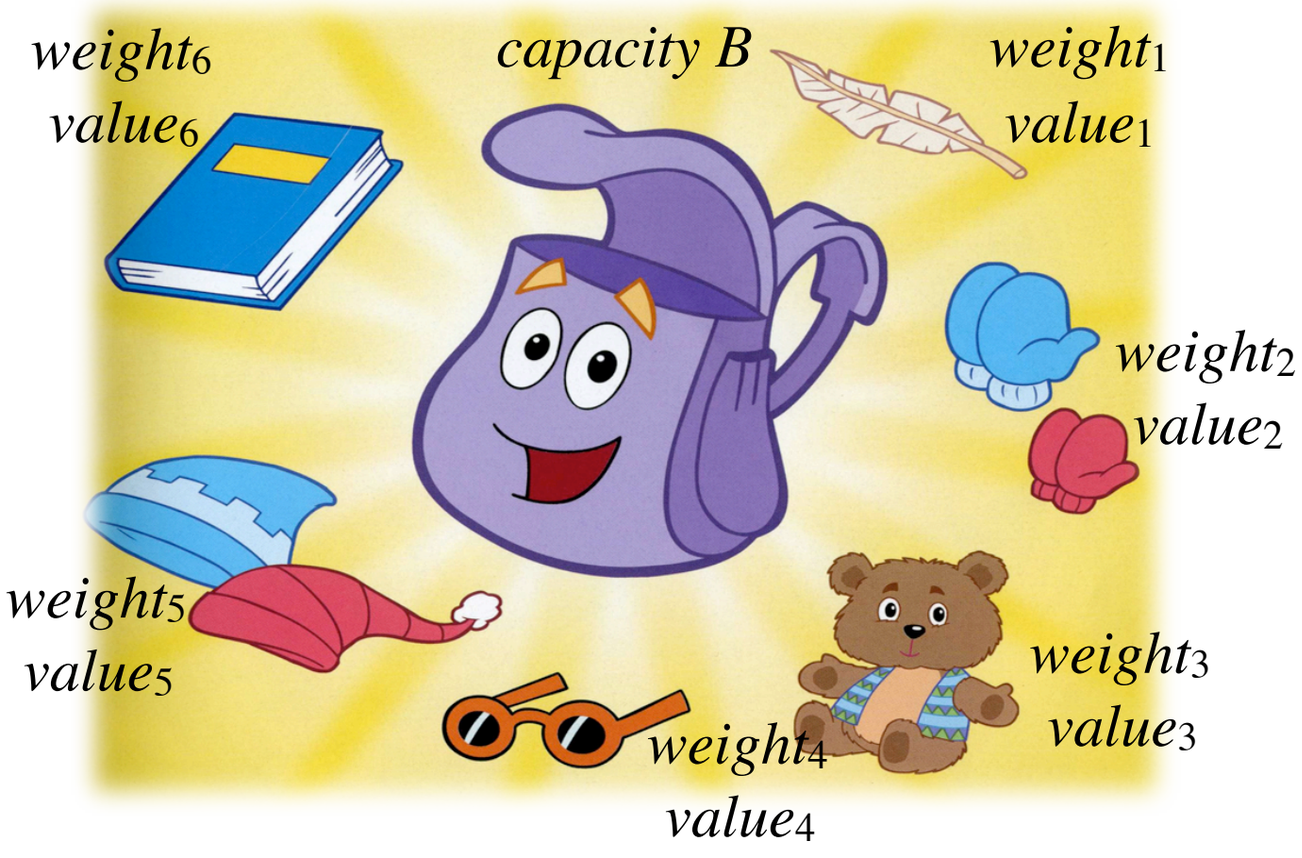
Instance: n items i=1,2, ..., n;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find a subset of items whose total weight is bounded by B and total value is maximized.

- 0-1 Knapsack problem
- one of Karp's 21 **NP**-complete problems



Polynomial Time

Instance: n items i=1,2, ..., n;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

time complexity: $O(nV)$ where $V = \sum_i v_i$

- polynomial-time Algorithm A:

  ∃constant c, ∀ input $x \in \{0,1\}^*$, $A(x)$ terminates in $|x|^c$ steps

  $|x|$ = length of input x (in *binary* code)

- pseudopolynomial-time Algorithm A:

  ∃constant c, ∀ input $x \in \{0,1\}^*$, $A(x)$ terminates in $|x|^c$ steps

  $|x|$ = length of input x (in *unary* code)

Dynamic Programming

Instance: n items i=1,2, ..., n;

weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}^+$; values $v_1, v_2, \ldots, v_n \in \mathbb{Z}^+$;

knapsack capacity $B \in \mathbb{Z}^+$;

Find a subset of items whose total weight is bounded by B and total value is maximized.

$A(i, v)$ = minimum total weight of $S \subseteq \{1,2, \ldots, i\}$ with total value *exactly* v

$A(i, v) = \min\{ A(i\text{-}1, v),\ A(i\text{-}1, v\text{-}v_i) + w_i \}$

$$A(1, v) = \begin{cases} w_1 & \text{if } v = v_1 \\ \infty & \text{otherwise} \end{cases}$$

knapsack: $\max\{v: A(n,v) \le B\}$

Dynamic programming time complexity $O(nV)$ where $V = \sum_i v_i$

Pseudo-Polynomial Time!

A Nearly Quadratic-Time FPTAS for Knapsack

Lin Chen*
Zhejiang University
Hangzhou, China
chenlin198662@zju.edu.cn

Jiayi Lian†
Zhejiang University
Hangzhou, China
jiayilian@zju.edu.cn

Yuchen Mao‡
Zhejiang University
Hangzhou, China
maoyc@zju.edu.cn

Guochuan Zhang§
Zhejiang University
Hangzhou, China
zgc@zju.edu.cn

ABSTRACT

We investigate the classic Knapsack problem and propose a fully polynomial-time approximation scheme (FPTAS) that runs in $\widetilde{O}(n+(1/\varepsilon)^2)$ time. Prior to our work, the best running time is $\widetilde{O}(n+(1/\varepsilon)^{11/5})$ [Deng, Jin, and Mao'23]. Our algorithm is the best possible (up to a polylogarithmic factor), as Knapsack has no $O((n+1/\varepsilon)^{2-\delta})$-time FPTAS for any constant $\delta > 0$, conditioned on the conjecture that $(\min,+)$-convolution has no truly subquadratic-time algorithm.

Knapsack is a fundamental problem in combinatorial optimization and belongs to Karp's 21 NP-complete problems [21]. Consequently, extensive effort has been devoted to developing approximation algorithms for Knapsack. Knapsack admits a fully polynomial-time approximation scheme (FPTAS), which is an algorithm that takes a precision parameter ε and produces a solution that is within a factor $1+\varepsilon$ from the optimum in time $\mathrm{poly}(n, 1/\varepsilon)$. Since the first FPTAS for Knapsack, there has been a long line of research on improving its running time, as summarized in Table 1.

STOC 2024: Proceedings of the 56th Annual ACM Symposium on Theory of Computing • Pages 283 - 294
https://doi.org/10.1145/3618260.3649730

Common subproblems

Finding the right subproblem takes creativity and experimentation. But there are a few standard choices that seem to arise repeatedly in dynamic programming.

i. The input is $x_1, x_2, \ldots, x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is therefore linear.

ii. The input is $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

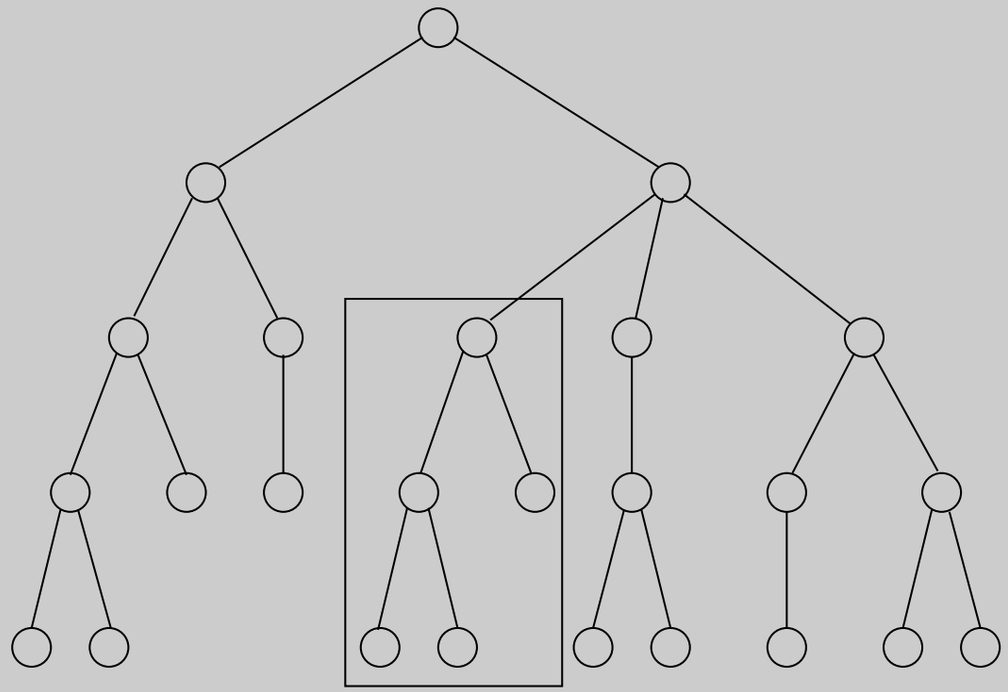
The number of subproblems is $O(mn)$.

iii. The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is $O(n^2)$.

iv. The input is a rooted tree. A subproblem is a rooted subtree.



If the tree has n nodes, how many subproblems are there?

Dynamic Programming

- Shortest path in DAGs
- Longest increasing subsequences
- Optimal binary search trees
- 0-1 knapsack
- Matrix multiplication
- Product assembly
- Shortest path, revisited

2. Ordering Matrix Multiplications

〖Example〗 Suppose we are to multiply 4 matrices

$$M_{1\,[\,10\times20\,]} * M_{2\,[\,20\times50\,]} * M_{3\,[\,50\times1\,]} * M_{4\,[\,1\times100\,]}\,.$$

If we multiply in the order

$$M_{1\,[\,10\times20\,]} * (M_{2\,[\,20\times50\,]} * (M_{3\,[\,50\times1\,]} * M_{4\,[\,1\times100\,]}))$$

Then the computing time is

$$50 \times 1 \times 100 + 20 \times 50 \times 100 + 10 \times 20 \times 100 = 125{,}000$$

If we multiply in the order

$$(M_{1\,[\,10\times20\,]} * (M_{2\,[\,20\times50\,]} * M_{3\,[\,50\times1\,]})) * M_{4\,[\,1\times100\,]}$$

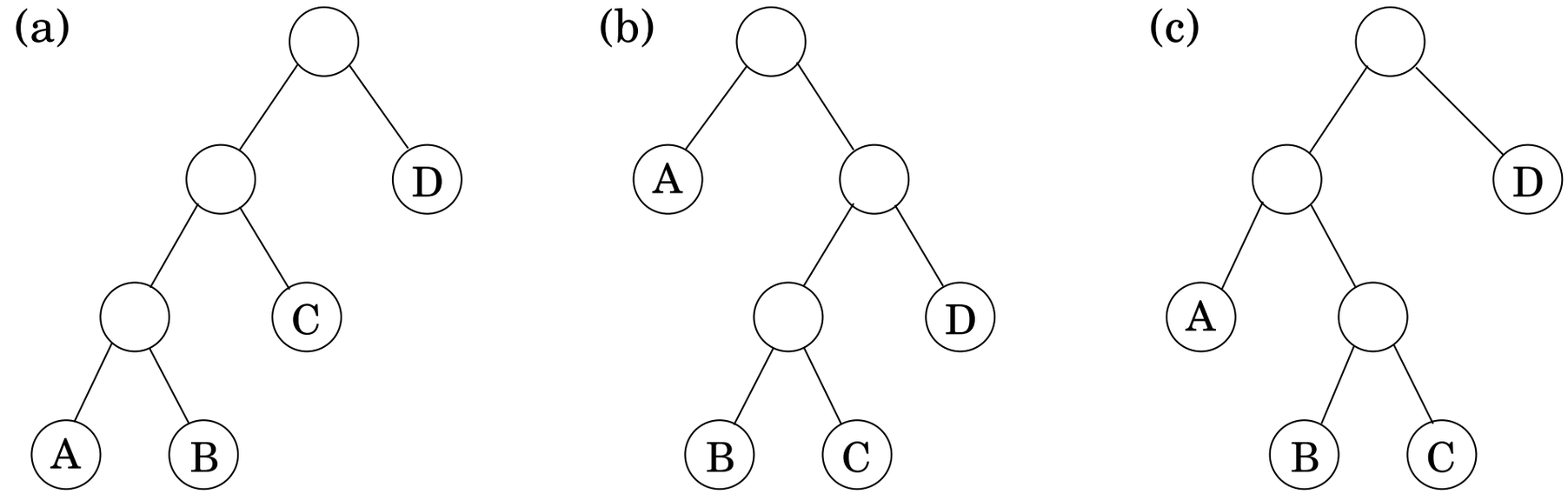
Then the computing time is

$$20 \times 50 \times 1 + 10 \times 20 \times 1 + 10 \times 1 \times 100 = 2{,}200$$

Problem: In which order can we compute the product of n matrices with minimal computing time?

No divide-and-conquer. Just use direct multiplication.

Figure 6.7 (a) $((A \times B) \times C) \times D$; (b) $A \times ((B \times C) \times D)$; (c) $(A \times (B \times C)) \times D$.



The sub-problems form a binary tree structure.

Let b_n = number of different ways to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_n$. Then we have $b_2 = 1, b_3 = 2, b_4 = 5, \cdots$

Let $M_{ij} = M_i \cdot \cdots \cdot M_j$. Then $M_{1n} = M_1 \cdot \cdots \cdot M_n = M_{1i} \cdot M_{i+1\ n}$

$$\Rightarrow \quad b_n = \sum_{i=1}^{n-1} b_i b_{n-i} \quad \text{wh}$$

$$b_n = O\left(\frac{4^n}{n\sqrt{n}}\right)$$

If $j - i = k$, then the only values M_{xy} required to compute M_{ij} satisfy $y - x < k$.

Suppose we are to multiply n matrices $M_1 * \cdots \cdots * M_n$ where M_i is an $r_{i-1} \times r_i$ matrix. Let m_{ij} be the cost of the optimal way to compute $M_i * \cdots \cdots * M_j$. Then we have the recurrence equations:

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le l < j} \{ m_{il} + m_{l+1\ j} + r_{i-1} r_l r_j \} & \text{if } j > i \end{cases}$$

```
/* r contains number of columns for each of the N matrices */
/* r[ 0 ] is the number of rows in matrix 1 */
/* Minimum number of multiplications is left in M[ 1 ][ N ] */
void OptMatrix( const long r[ ], int N, TwoDimArray M )
{   int  i, j, k, L;
    long  ThisM;
    for( i = 1; i <= N; i++ )   M[ i ][ i ] = 0;
    for( k = 1; k < N; k++ ) /* k = j - i */
        for( i = 1; i <= N - k; i++ ) { /* For each position */
            j = i + k;    M[ i ][ j ] = Infinity;
            for( L = i; L < j; L++ ) {
                ThisM = M[ i ][ L ] + M[ L + 1 ][ j ]
                        + r[ i - 1 ] * r[ L ] * r[ j ];
                if ( ThisM < M[ i ][ j ] )  /* Update min */
                    M[ i ][ j ] = ThisM;
            } /* end for-L */
        } /* end for-Left */
}
```

$$\begin{matrix} m_{1,N} & & & & \\ m_{1,N-1} & m_{2,N} & & & \\ \vdots & \vdots & \ddots & & \\ m_{1,2} & m_{2,3} & \cdots & m_{N-1,N} & \\ m_{1,1} & m_{2,2} & \cdots & m_{N-1,N-1} & m_{N,N} \end{matrix}$$

$$T(N) = \mathrm{O}(N^3)$$

To record the ordering please refer to Weiss Figure 10.46 on p.388

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le l < j} \{ m_{il} + m_{l+1\,j} + r_{i-1} r_l r_j \} & \text{if } j > i \end{cases}$$

```
MATRIX-CHAIN-ORDER(p, n)
 1  let m[1:n, 1:n] and s[1:n − 1, 2:n] be new tables
 2  for i = 1 to n                          // chain length 1
 3      m[i, i] = 0
 4  for l = 2 to n                          // l is the chain length
 5      for i = 1 to n − l + 1              // chain begins at A_i
 6          j = i + l − 1                   // chain ends at A_j
 7          m[i, j] = ∞
 8          for k = i to j − 1              // try A_{i:k} A_{k+1:j}
 9              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10              if q < m[i, j]
11                  m[i, j] = q             // remember this cost
12                  s[i, j] = k             // remember this index
13  return m and s
```

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

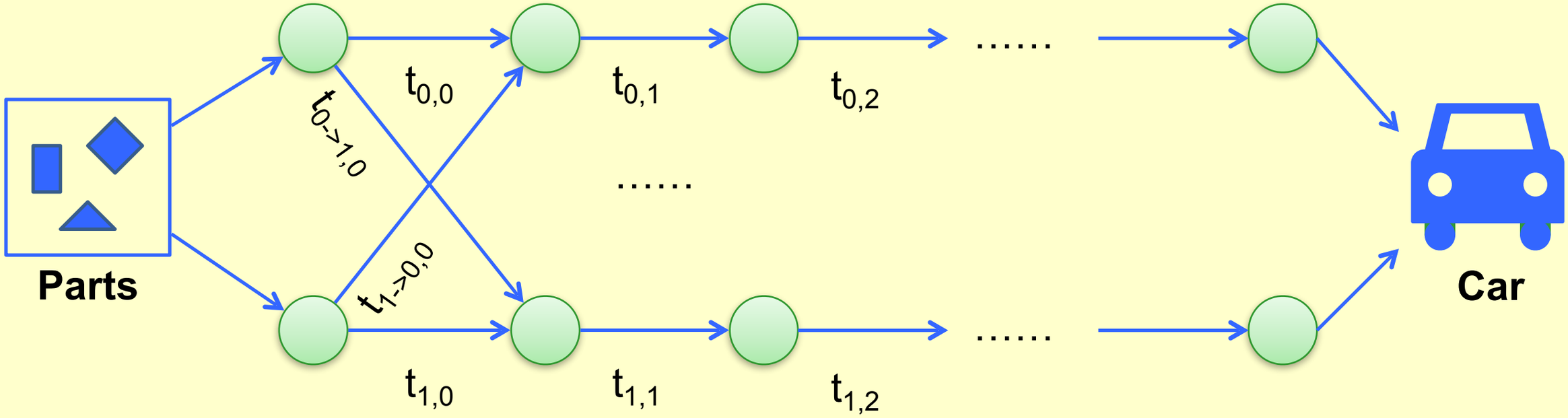
CLRS (4th edition) pp. 378, 381

Dynamic Programming

- Shortest path in DAGs
- Longest increasing subsequences
- Optimal binary search trees
- 0-1 knapsack
- Matrix multiplication
- Product assembly
- Shortest path, revisited

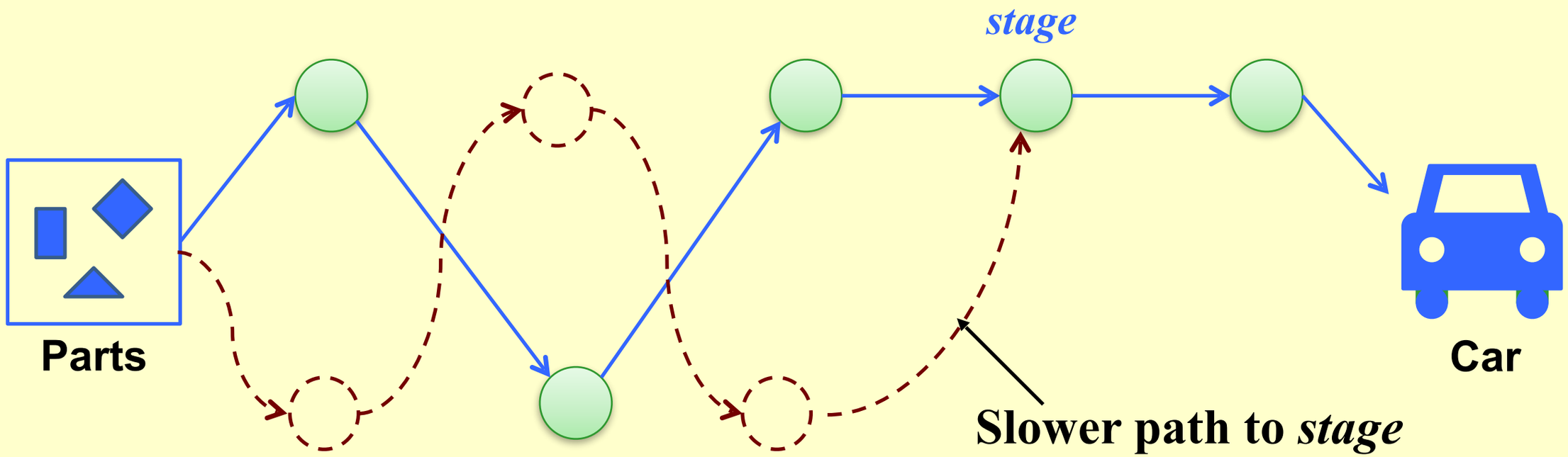
5. Product Assembly

- Two assembly lines for the same car
- Different technology (time) for each stage
- One can change lines between stages
- Minimize the total assembly time



Exhaustive search gives O(2^N) time + O(N) space

☞ Characterize an optimal solution

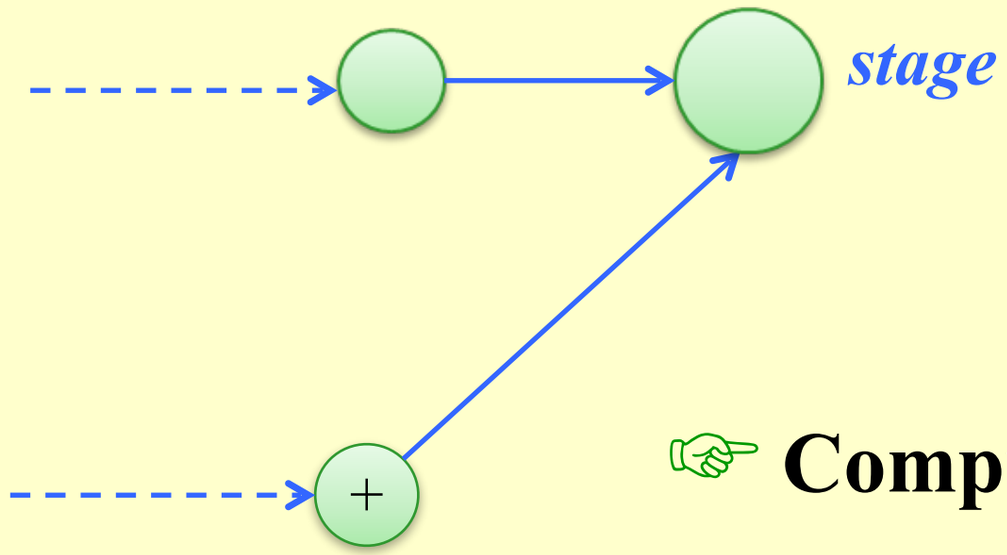


Optimal solution

☝ **An optimal solution contains an optimal solution of a sub-problem!**

☞ Recursively define the optimal values

An optimal path to *stage* is based on an optimal path to *stage*–1



☞ Compute the values in some order

O(*N*) time + O(*N*) space

```
f[0][0]=0;  f[1][0]=0;
for (stage=1; stage<=n; stage++){
   for (line=0; line<=1; line++){
   f[line][stage] =
       f[  line][stage-1] + t_process[  line][stage-1],


   }
}
Solution = min(f[0][n],f[1][n]);
```

☞ Reconstruct the solving strategy

```
f[0][0]=0;
f[1][0]=0;
for(stage=1; stage<=n; stage++){
  for(line=0; line<=1; line++){
    f_stay = f[  line][stage-1] + t_process[  line][stage-1];
    f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];
    if (f_stay<f_move){
      f[line][stage] = f_stay;

    }
    else {
      f[line][stage] = f_move;

    }
  }
}
```

```
line = f[0][n]<f[1][n]?0:1;
for(stage=n; stage>0; stage--){
  plan[stage] = line;
  line = L[line][stage];
}
```

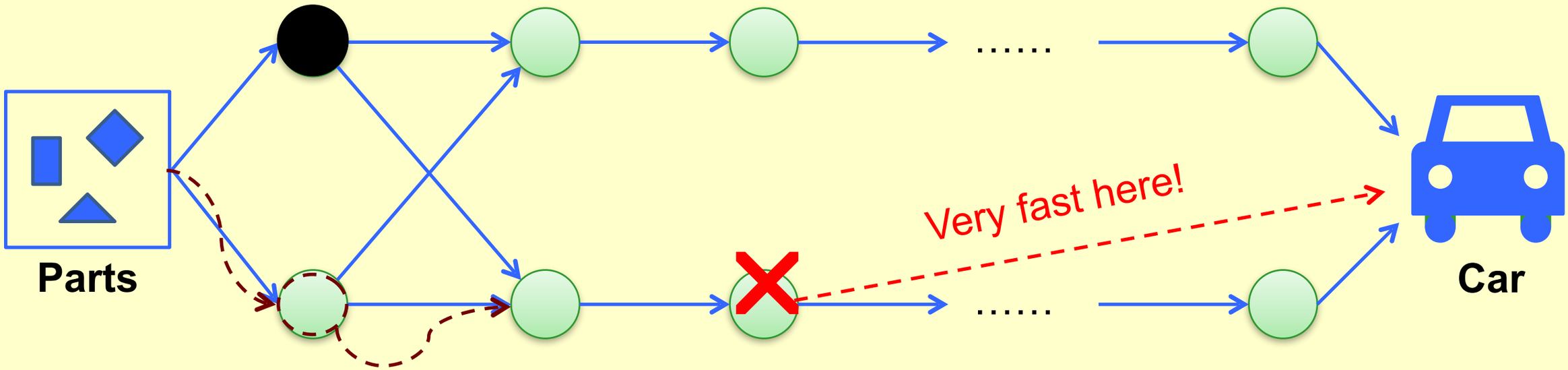
Elements of DP:

- Optimal substructure
- Overlapping sub-problems

If sub-problems do not overlap…
It becomes sort of pointless

Discussion 11:
When *can't* we apply dynamic programming?

History-dependency



Dynamic Programming

- Shortest path in DAGs
- Longest increasing subsequences
- Optimal binary search trees
- 0-1 knapsack
- Matrix multiplication
- Product assembly
- Shortest path, revisited

4. All-Pairs Shortest Path

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

Method 1 Use **single-source algorithm** for |V| times.

$T = O(|V|^3)$ – works fast on sparse graph.

Method 2 Define

$D^k[\,i\,][\,j\,] = \min\{$ length of path $i \rightarrow \{\, l \leq k \,\} \rightarrow j \,\}$

and $D^{-1}[\,i\,][\,j\,] = \text{Cost}[\,i\,][\,j\,]$. Then the length of the shortest path from i to j is $D^{N-1}[\,i\,][\,j\,]$.

Algorithm

Start from D^{-1} and successively generate $D^0, D^1, \ldots, D^{N-1}$. If D^{k-1} is done, then either

① $k \notin$ the shortest path $i \rightarrow \{\, l \leq k \,\} \rightarrow j \;\Rightarrow\; D^k = D^{k-1}$;

or

② $k \in$ the shortest path $i \rightarrow \{\, l \leq k \,\} \rightarrow j$

$=$ { the S.P. from i to k } $\cup$ {the S.P. from k to j }

$\Rightarrow\; D^k[\,i\,][\,j\,] = D^{k-1}[\,i\,][\,k\,] + D^{k-1}[\,k\,][\,j\,]$

$$\therefore\quad D^k[i][j] = \min\{D^{k-1}[i][j],\ D^{k-1}[i][k] + D^{k-1}[k][j]\},\ k \geq 0$$

```
/* A[ ] contains the adjacency matrix with A[ i ][ i ] = 0 */
/* D[ ] contains the values of the shortest path */
/* N is the number of vertices */
/* A negative cycle exists iff D[ i ][ i ] < 0 */
void AllPairs( TwoDimArray A, TwoDimArray D, int N )
{   int  i, j, k;
    for ( i = 0; i < N; i++ )     /* Initialize D */
        for( j = 0; j < N; j++ )
                D[ i ][ j ] = A[ i ][ j ];
    for( k = 0; k < N; k++ )
        for( i = 0; i < N; i++ )
            for( j = 0; j < N; j++ )
                if( D[ i ][ k ] + D[ k ][ j ] < D[ i ][ j ] )
                    /* Update shortest path */
                    D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ];
}
```

Works if there are negative edge costs, but no negative-cost cycles.

$T(N) = O(N^3)$, but faster in a *dense* graph.

To record the paths please refer to Weiss Figure 10.53 on p.393

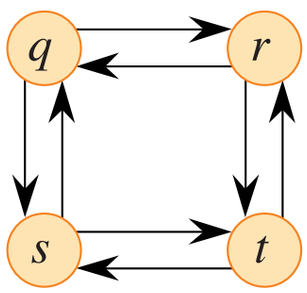


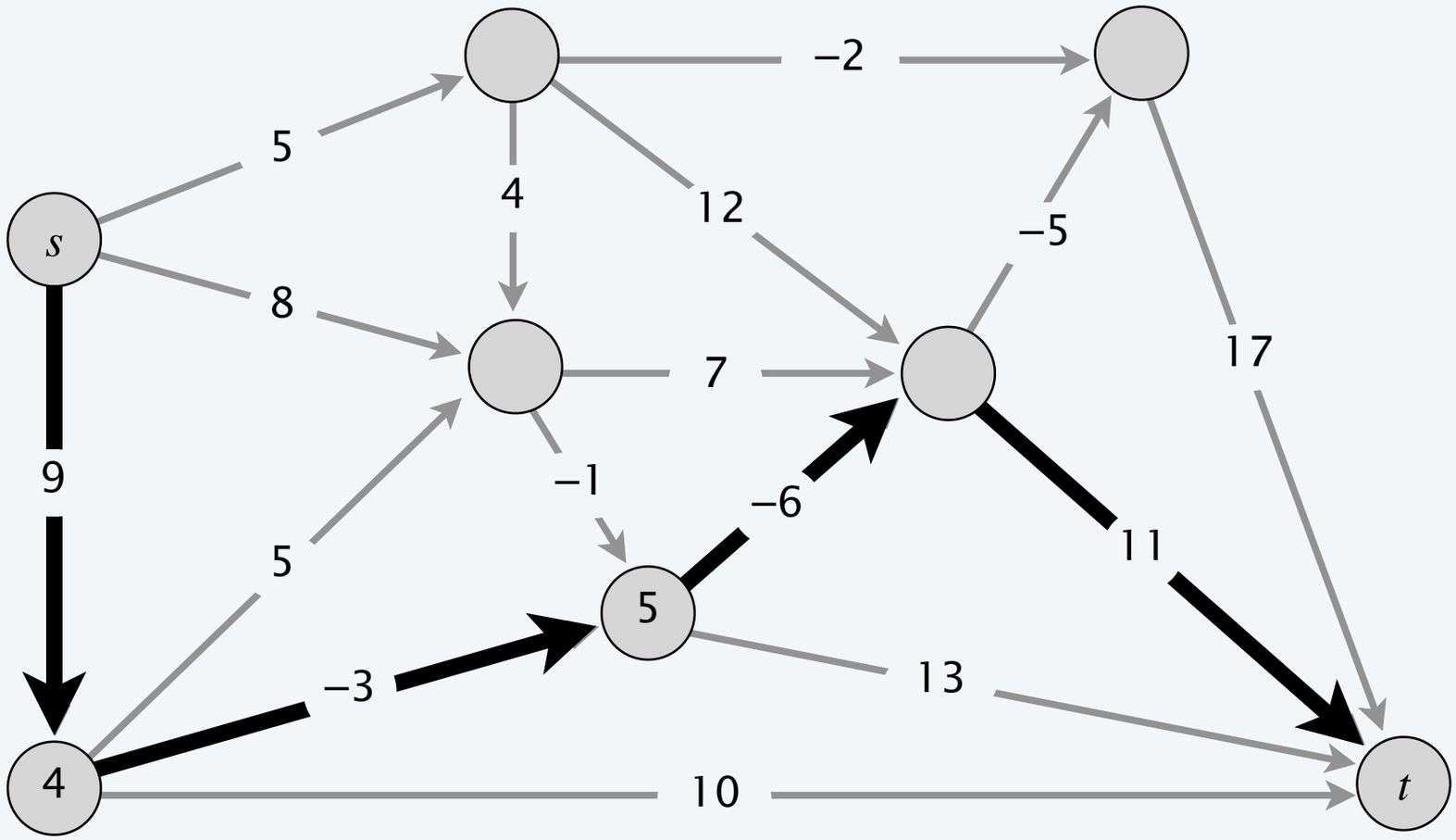
Figure 14.6 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \to r \to t$ is a longest simple path from q to t, but the subpath $q \to r$ is not a longest simple path from q to r, nor is the subpath $r \to t$ a longest simple path from r to t.

To use DP, it is essential to verify whether the substructure optimal condition really holds.
Read CLRS (4th edition) pp. 386

Shortest paths with negative weights

Shortest-path problem. Given a digraph $G = (V, E)$, with arbitrary edge lengths ℓ_{vw}, find shortest path from source node s to destination node t.

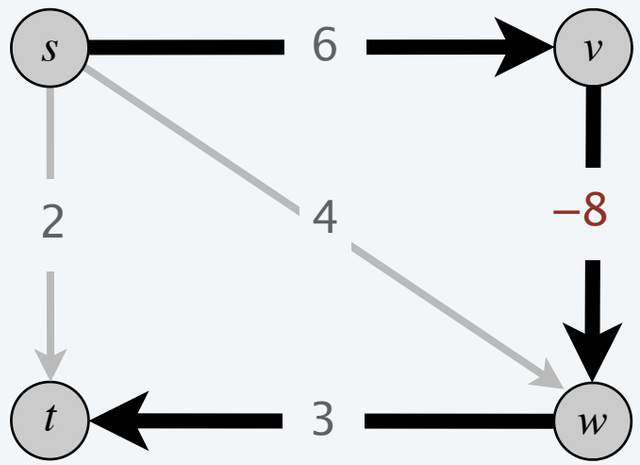
assume there exists a path from every node to t



length of shortest $s \rightsquigarrow t$ path = 9 − 3 − 6 + 11 = 11

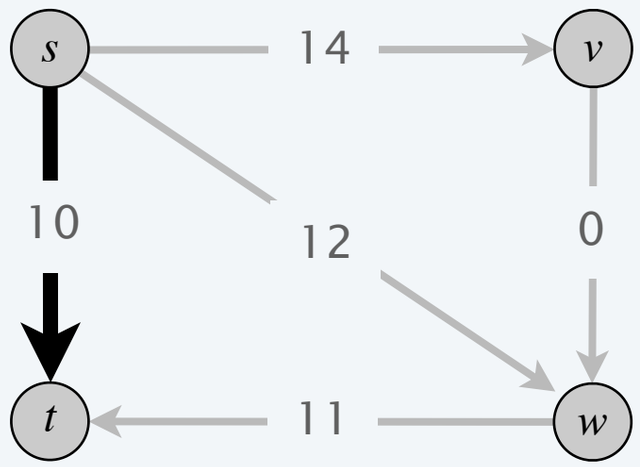
Shortest paths with negative weights: failed attempts

Dijkstra. May not produce shortest paths when edge lengths are negative.



Dijkstra selects the vertices in the order s, t, w, v
But shortest path from s to t is $s \rightarrow v \rightarrow w \rightarrow t$.

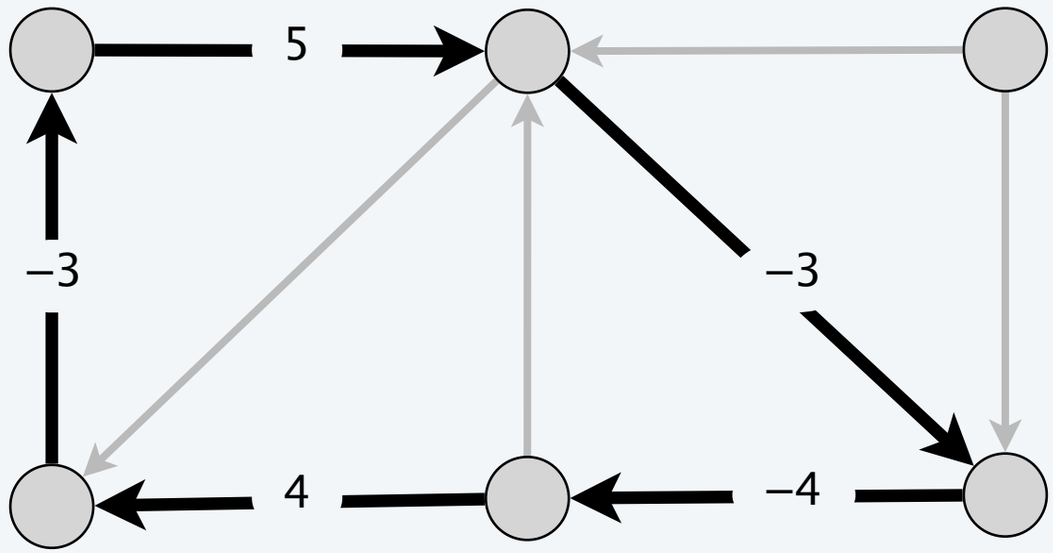
Reweighting. Adding a constant to every edge length does not necessarily make Dijkstra's algorithm produce shortest paths.



Adding 8 to each edge weight changes the shortest path from $s \rightarrow v \rightarrow w \rightarrow t$ to $s \rightarrow t$.

Negative cycles

Def. A negative cycle is a directed cycle for which the sum of its edge lengths is negative.

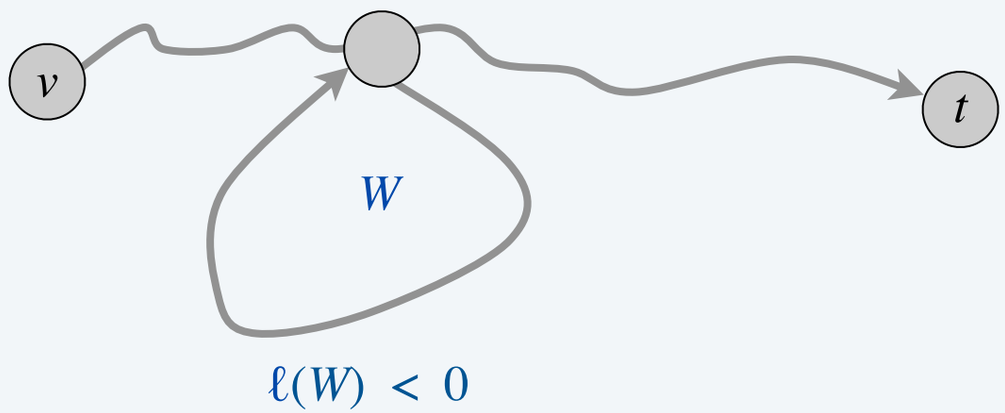


a negative cycle W : $\ell(W) = \sum_{e \in W} \ell_e < 0$

Shortest paths and negative cycles

Lemma 1. If some $v \rightsquigarrow t$ path contains a negative cycle, then there does not exist a shortest $v \rightsquigarrow t$ path.

Pf. If there exists such a cycle W, then can build a $v \rightsquigarrow t$ path of arbitrarily negative length by detouring around W as many times as desired. ▪

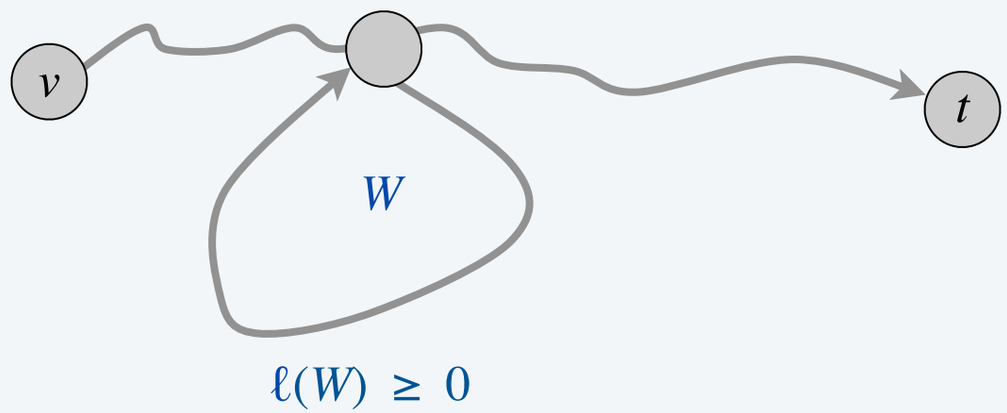


Shortest paths and negative cycles

Lemma 2. If G has no negative cycles, then there exists a shortest $v \rightsquigarrow t$ path that is simple (and has $\leq n - 1$ edges).

Pf.

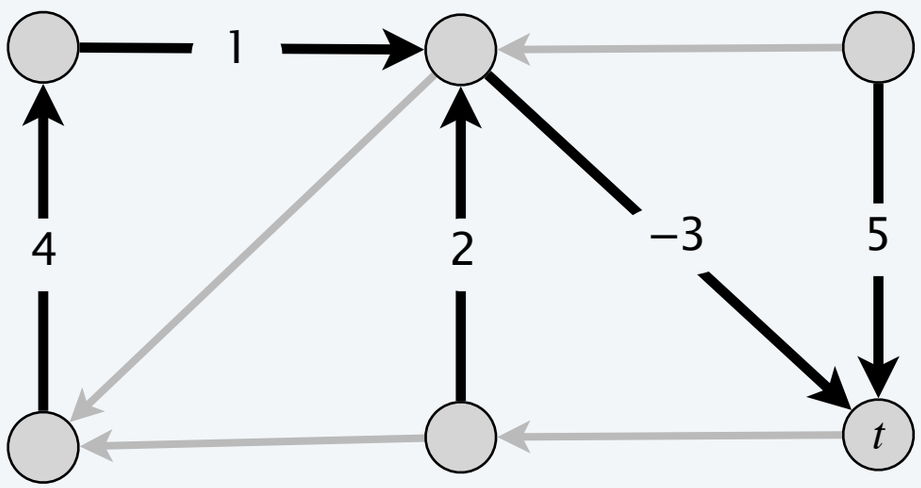
- Among all shortest $v \rightsquigarrow t$ paths, consider one that uses the fewest edges.
- If that path P contains a directed cycle W, can remove the portion of P corresponding to W without increasing its length. ▪



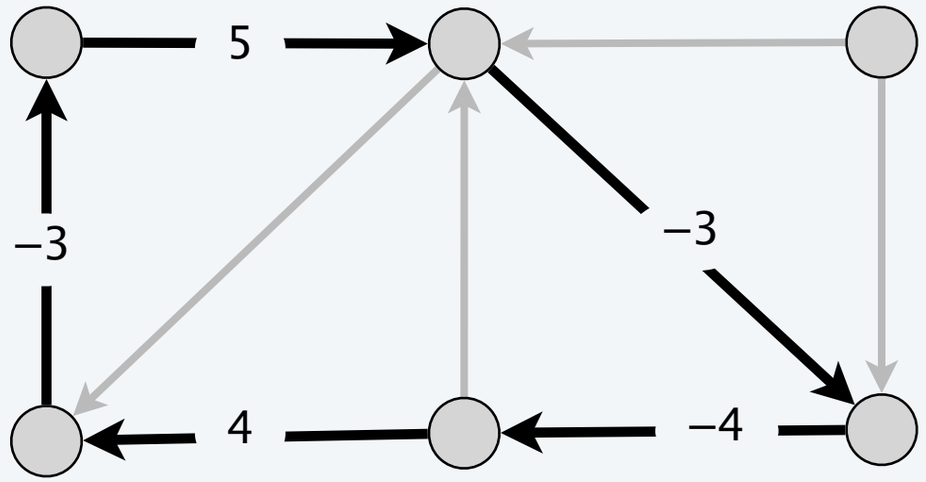
Shortest-paths and negative-cycle problems

Single-destination shortest-paths problem. Given a digraph $G = (V, E)$ with edge lengths ℓ_{vw} (but no negative cycles) and a distinguished node t, find a shortest $v \rightsquigarrow t$ path for every node v.

Negative-cycle problem. Given a digraph $G = (V, E)$ with edge lengths ℓ_{vw}, find a negative cycle (if one exists).



shortest-paths tree



negative cycle

Dynamic programming: quiz 5

Which subproblems to find shortest $v \rightsquigarrow t$ paths for every node v?

A. $OPT(i, v)$ = length of shortest $v \rightsquigarrow t$ path that uses exactly i edges.

B. $OPT(i, v)$ = length of shortest $v \rightsquigarrow t$ path that uses at most i edges.

C. Neither A nor B.

Shortest paths with negative weights: dynamic programming

Def. $OPT(i, v)$ = length of shortest $v \rightsquigarrow t$ path that uses $\leq i$ edges.

Goal. $OPT(n-1, v)$ for each v.

by Lemma 2, if no negative cycles, there exists a shortest $v \rightsquigarrow t$ path that is simple

Case 1. Shortest $v \rightsquigarrow t$ path uses $\leq i-1$ edges.

- $OPT(i, v) = \mathrm{OPT}(i-1, v)$.

optimal substructure property (proof via exchange argument)

Case 2. Shortest $v \rightsquigarrow t$ path uses exactly i edges.

- if (v, w) is first edge in shortest such $v \rightsquigarrow t$ path, incur a cost of ℓ_{vw}.
- Then, select best $w \rightsquigarrow t$ path using $\leq i-1$ edges.

Bellman equation.

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ \infty & \text{if } i = 0 \text{ and } v \neq t \\ \min\left\{ OPT(i-1, v),\ \min\limits_{(v,w) \in E} \{OPT(i-1, w) + \ell_{vw}\} \right\} & \text{if } i > 0 \end{cases}$$

Shortest paths with negative weights: implementation

SHORTEST-PATHS(V, E, ℓ, t)

FOREACH node $v \in V$:

$M[0, v] \leftarrow \infty$.

$M[0, t] \leftarrow 0$.

FOR i = 1 TO $n - 1$

FOREACH node $v \in V$:

$M[i, v] \leftarrow M[i-1, v]$.

FOREACH edge $(v, w) \in E$:

$M[i, v] \leftarrow \min \{ M[i, v],\ M[i-1, w] + \ell_{vw} \}$.

Shortest paths with negative weights: implementation

Theorem 1. Given a digraph $G = (V, E)$ with no negative cycles, the DP algorithm computes the length of a shortest $v \rightsquigarrow t$ path for every node v in $\Theta(mn)$ time and $\Theta(n^2)$ space.

Pf.

- Table requires $\Theta(n^2)$ space.
- Each iteration i takes $\Theta(m)$ time since we examine each edge once. ▪

Finding the shortest paths.

- Approach 1: Maintain $successor[i, v]$ that points to next node on a shortest $v \rightsquigarrow t$ path using $\leq i$ edges.
- Approach 2: Compute optimal lengths $M[i, v]$ and consider only edges with $M[i, v] = M[i-1, w] + \ell_{vw}$. Any directed path in this subgraph is a shortest path.

Dynamic programming: quiz 6

It is easy to modify the DP algorithm for shortest paths to…

A. Compute lengths of shortest paths in $O(mn)$ time and $O(m + n)$ space.

B. Compute shortest paths in $O(mn)$ time and $O(m + n)$ space.

C. Both A and B.

D. Neither A nor B.

Shortest paths with negative weights: practical improvements

Space optimization. Maintain two 1D arrays (instead of 2D array).

- $d[v]$ = length of a shortest $v \rightsquigarrow t$ path that we have found so far.
- $successor[v]$ = next node on a $v \rightsquigarrow t$ path.

Performance optimization. If $d[w]$ was not updated in iteration $i - 1$, then no reason to consider edges entering w in iteration i.

Bellman–Ford–Moore: efficient implementation

BELLMAN–FORD–MOORE(V, E, c, t)

FOREACH node $v \in V$:

  $d[v] \leftarrow \infty$.

  $successor[v] \leftarrow null$.

$d[t] \leftarrow 0$.

FOR $i = 1$ TO $n - 1$

  FOREACH node $w \in V$:

    IF ($d[w]$ was updated in previous pass)

      FOREACH edge $(v, w) \in E$:

        IF ($d[v] > d[w] + \ell_{vw}$)

          $d[v] \leftarrow d[w] + \ell_{vw}$.

          $successor[v] \leftarrow w$.

  IF (no $d[\cdot]$ value changed in pass i) STOP.

pass i
$O(m)$ time

Dynamic programming: quiz 7

Which properties must hold after pass i of Bellman–Ford–Moore?

A. $d[v]$ = length of a shortest $v \rightsquigarrow t$ path using $\leq i$ edges.

B. $d[v]$ = length of a shortest $v \rightsquigarrow t$ path using exactly i edges.

C. Both A and B.

D. Neither A nor B.

Bellman–Ford–Moore: analysis

Lemma 3. For each node v : $d[v]$ is the length of some $v \rightsquigarrow t$ path.

Lemma 4. For each node v : $d[v]$ is monotone non-increasing.

Lemma 5. After pass i, $d[v] \leq$ length of a shortest $v \rightsquigarrow t$ path using $\leq i$ edges.

Pf. [by induction on i]

- Base case: $i = 0$.
- Assume true after pass i.
- Let P be any $v \rightsquigarrow t$ path with $\leq i + 1$ edges.
- Let (v, w) be first edge in P and let P' be subpath from w to t.
- By inductive hypothesis, at the end of pass i, $d[w] \leq \ell(P')$ because P' is a $w \rightsquigarrow t$ path with $\leq i$ edges.
- After considering edge (v, w) in pass $i + 1$:

and by Lemma 4, $d[w]$ does not increase

$$\begin{aligned} d[v] &\leq \ell_{vw} + d[w] \\ &\leq \ell_{vw} + \ell(P') \\ &= \ell(P) \quad \blacksquare \end{aligned}$$

and by Lemma 4, $d[v]$ does not increase

Bellman–Ford–Moore: analysis

Theorem 2. Assuming no negative cycles, Bellman–Ford–Moore computes the lengths of the shortest $v \rightsquigarrow t$ paths in $O(mn)$ time and $\Theta(n)$ extra space.

Pf. Lemma 2 + Lemma 5. ▪

shortest path exists and has at most $n-1$ edges

after i passes, $d[v] \leq$ length of shortest path that uses $\leq i$ edges

Remark. Bellman–Ford–Moore is typically faster in practice.

- Edge (v, w) considered in pass $i+1$ only if $d[w]$ updated in pass i.
- If shortest path has k edges, then algorithm finds it after $\leq k$ passes.

Dynamic programming: quiz 8

Assuming no negative cycles, which properties must hold throughout Bellman–Ford–Moore?

A. Following $successor[v]$ pointers gives a directed $v \rightsquigarrow t$ path.

B. If following $successor[v]$ pointers gives a directed $v \rightsquigarrow t$ path, then the length of that $v \rightsquigarrow t$ path is $d[v]$.

C. Both A and B.

D. Neither A nor B.

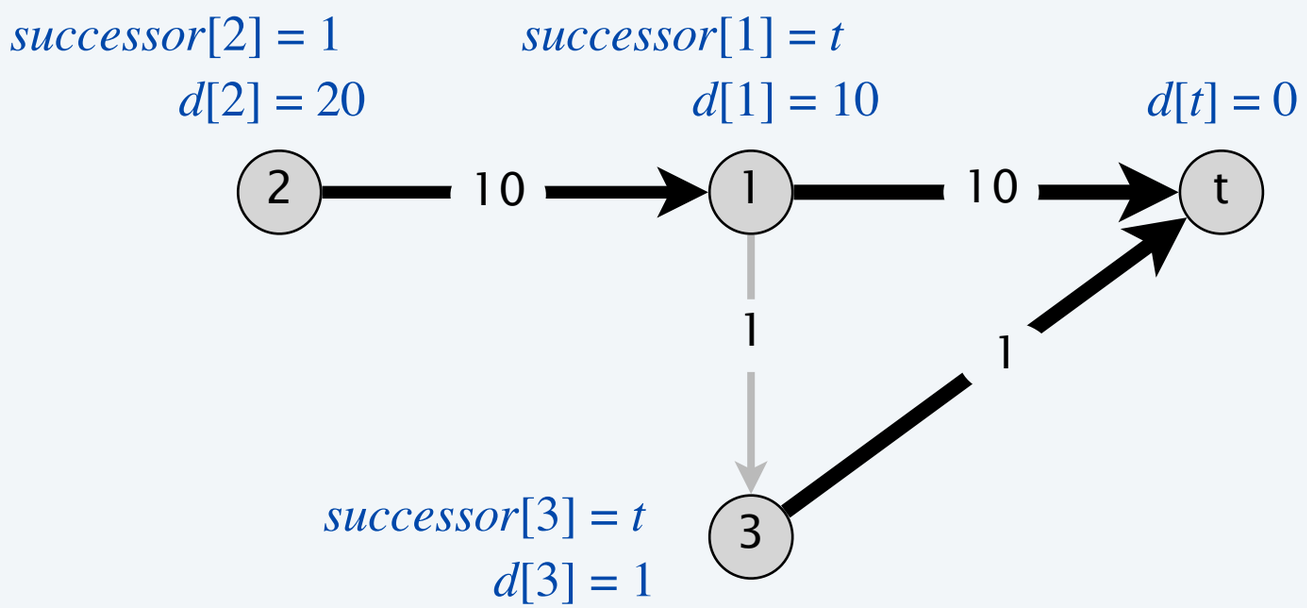
Bellman–Ford–Moore: analysis

Claim. ~~Throughout Bellman–Ford–Moore, following the *successor*[*v*] pointers gives a directed path from *v* to *t* of length *d*[*v*].~~

Counterexample. Claim is false!

- Length of successor $v \rightsquigarrow t$ path may be strictly shorter than $d[v]$.

consider nodes in order: t, 1, 2, 3



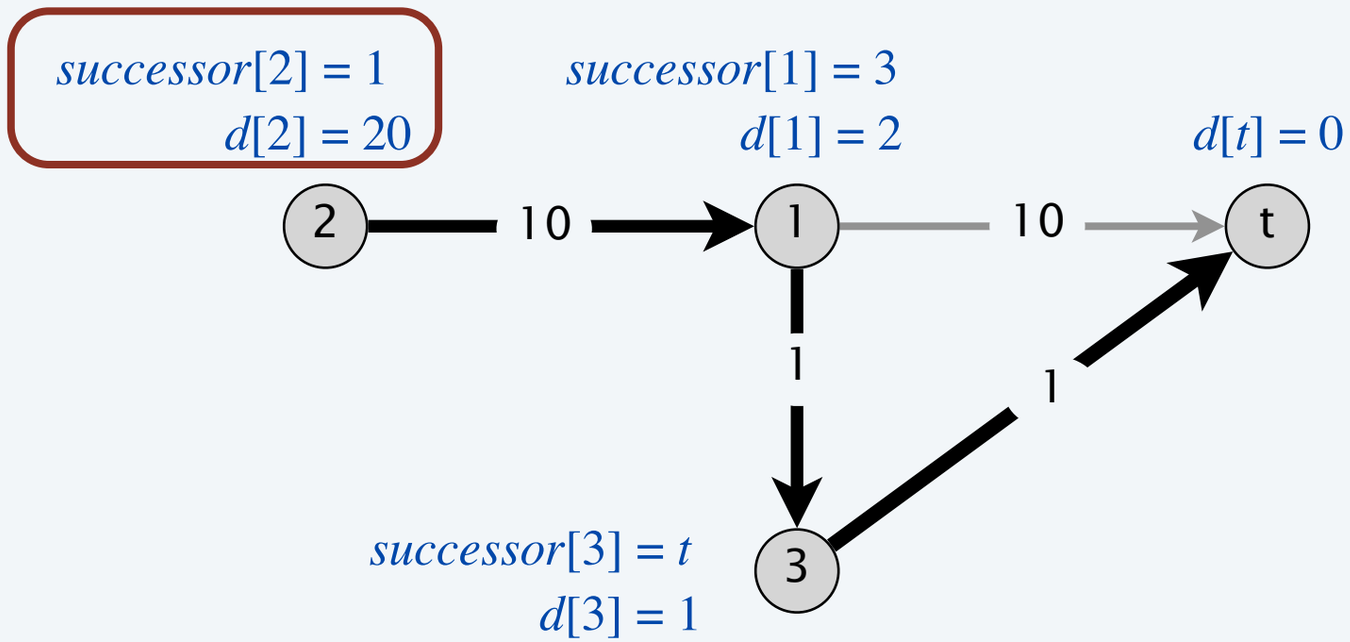
Bellman–Ford–Moore: analysis

Claim. ~~Throughout Bellman–Ford–Moore, following the *successor*[*v*] pointers gives a directed path from *v* to *t* of length *d*[*v*].~~

Counterexample. Claim is false!

- Length of successor $v \rightsquigarrow t$ path may be strictly shorter than $d[v]$.

consider nodes in order: t, 1, 2, 3



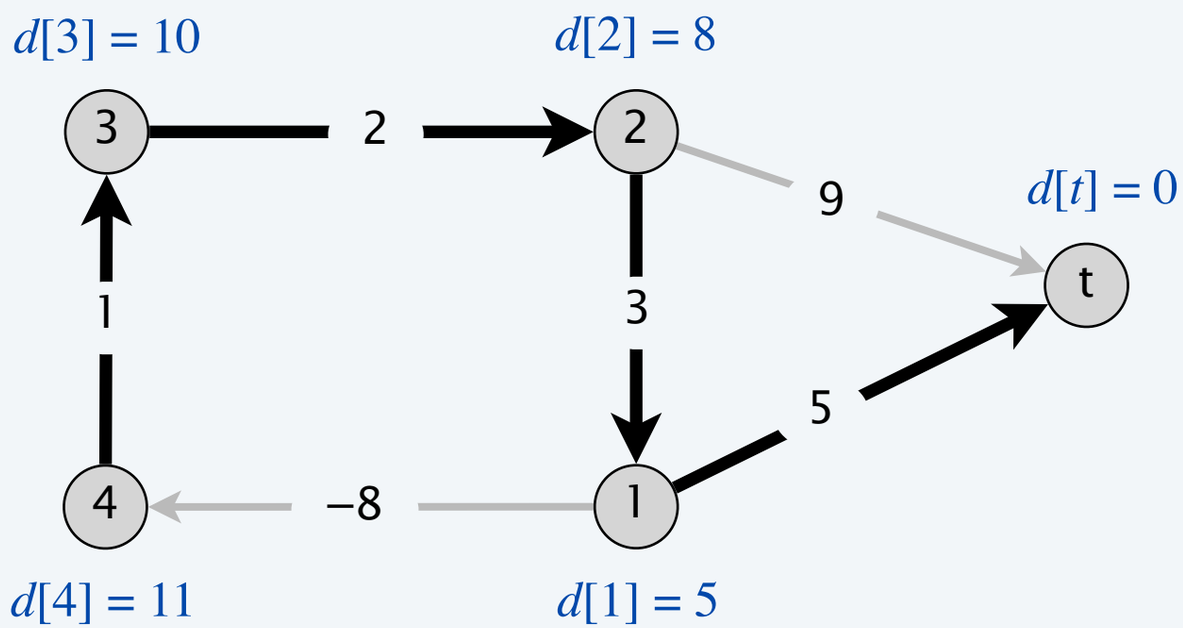
Bellman–Ford–Moore: analysis

Claim. ~~Throughout Bellman–Ford–Moore, following the *successor*[*v*] pointers gives a directed path from *v* to *t* of length *d*[*v*].~~

Counterexample. Claim is false!

- Length of successor $v \rightsquigarrow t$ path may be strictly shorter than $d[v]$.
- If negative cycle, successor graph may have directed cycles.

consider nodes in order: t, 1, 2, 3, 4



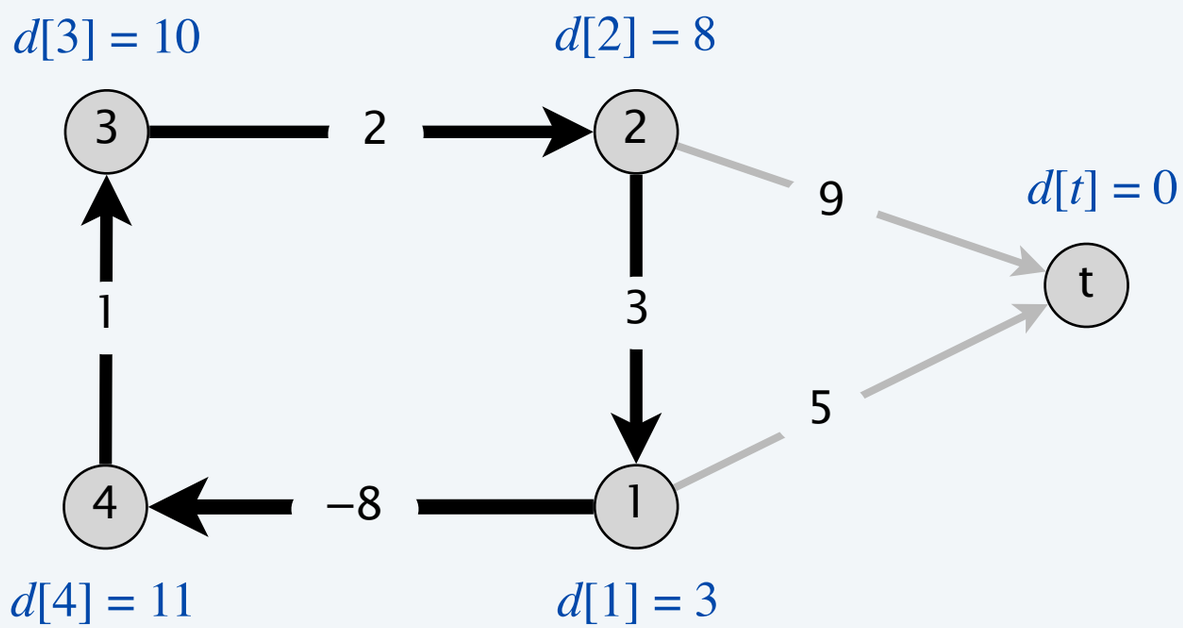
Bellman–Ford–Moore: analysis

Claim. ~~Throughout Bellman–Ford–Moore, following the *successor*[*v*] pointers gives a directed path from *v* to *t* of length *d*[*v*].~~

Counterexample. Claim is false!

- Length of successor $v \rightsquigarrow t$ path may be strictly shorter than $d[v]$.
- If negative cycle, successor graph may have directed cycles.

consider nodes in order: t, 1, 2, 3, 4



Bellman–Ford–Moore: finding the shortest paths

Lemma 6. Any directed cycle W in the successor graph is a negative cycle.

Pf.

- If $successor[v] = w$, we must have $d[v] \geq d[w] + \ell_{vw}$.
  (LHS and RHS are equal when $successor[v]$ is set; $d[w]$ can only decrease; $d[v]$ decreases only when $successor[v]$ is reset)
- Let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$ be the sequence of nodes in a directed cycle W.
- Assume that (v_k, v_1) is the last edge in W added to the successor graph.
- Just prior to that:

$$\begin{aligned}
d[v_1] &\geq d[v_2] + \ell(v_1, v_2) \\
d[v_2] &\geq d[v_3] + \ell(v_2, v_3) \\
&\vdots \quad\quad \vdots \quad\quad \vdots \\
d[v_{k-1}] &\geq d[v_k] + \ell(v_{k-1}, v_k) \\
d[v_k] &> d[v_1] + \ell(v_k, v_1)
\end{aligned}$$

holds with strict inequality since we are updating $d[v_k]$

- Adding inequalities yields $\ell(v_1, v_2) + \ell(v_2, v_3) + \ldots + \ell(v_{k-1}, v_k) + \ell(v_k, v_1) < 0$. ▪

W is a negative cycle

Bellman–Ford–Moore: finding the shortest paths

Theorem 3. Assuming no negative cycles, Bellman–Ford–Moore finds shortest $v \rightsquigarrow t$ paths for every node v in $O(mn)$ time and $\Theta(n)$ extra space.

Pf.

- The successor graph cannot have a directed cycle. [Lemma 6]
- Thus, following the successor pointers from v yields a directed path to t.
- Let $v = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = t$ be the nodes along this path P.
- Upon termination, if $successor[v] = w$, we must have $d[v] = d[w] + \ell_{vw}$.
  (LHS and RHS are equal when $successor[v]$ is set; $d[\cdot]$ did not change)
- Thus,

$$\begin{aligned} d[v_1] &= d[v_2] + \ell(v_1, v_2) \\ d[v_2] &= d[v_3] + \ell(v_2, v_3) \\ &\vdots \quad\quad \vdots \quad\quad \vdots \\ d[v_{k-1}] &= d[v_k] + \ell(v_{k-1}, v_k) \end{aligned}$$

since algorithm terminated

- Adding equations yields $d[v] = d[t] + \ell(v_1, v_2) + \ell(v_2, v_3) + \ldots + \ell(v_{k-1}, v_k)$. ∎

min length of any $v \rightsquigarrow t$ path (Theorem 2)

0

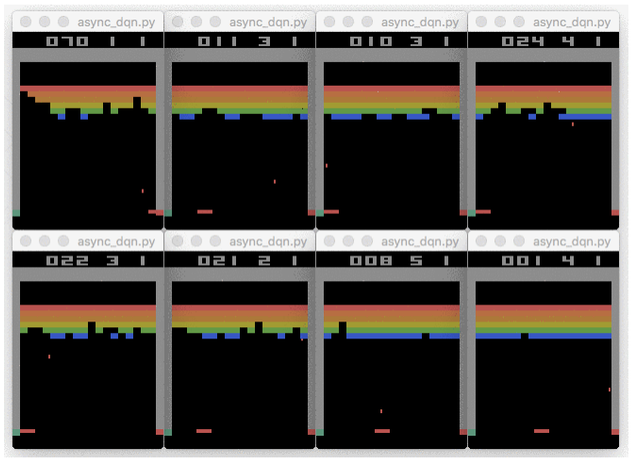
length of path P

Single-source shortest paths with negative weights

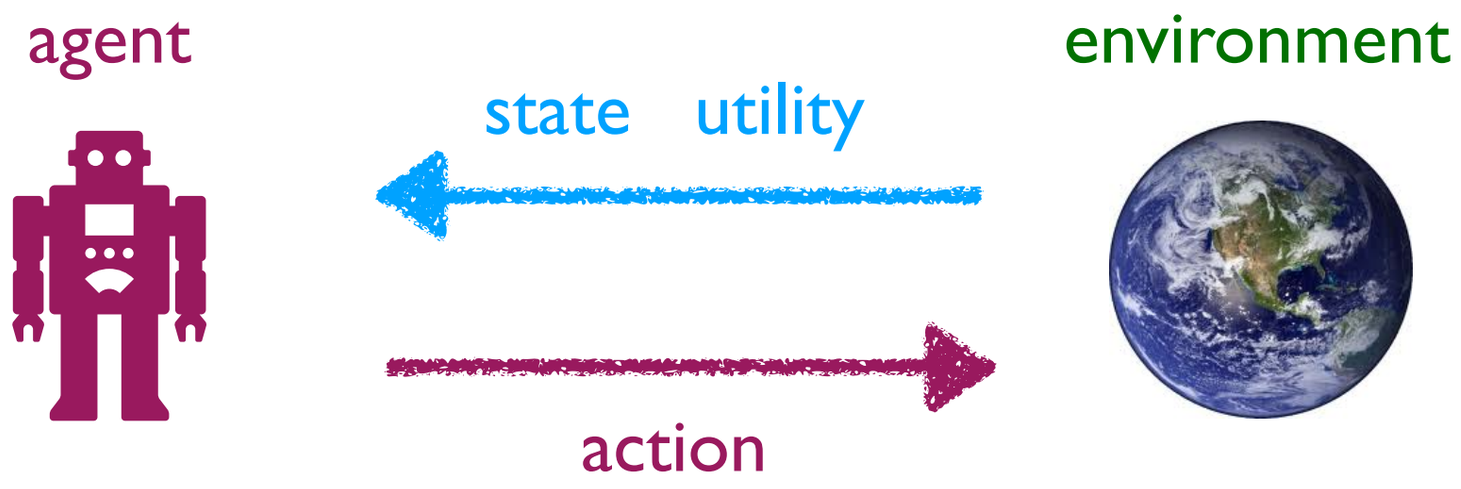
year	worst case	discovered by
1955	$O(n^4)$	Shimbel
1956	$O(m\, n^2\, W)$	Ford
1958	$O(m\, n)$	Bellman, Moore
1983	$O(n^{3/4}\, m \log W)$	Gabow
1989	$O(m\, n^{1/2} \log(nW))$	Gabow–Tarjan
1993	$O(m\, n^{1/2} \log W)$	Goldberg
2005	$O(n^{2.38}\, W)$	Sankowsi, Yuster–Zwick
2016	$\tilde{O}(n^{10/7} \log W)$	Cohen–Mądry–Sankowski–Vladu
20xx		

single-source shortest paths with weights between –W and W

Decision Making

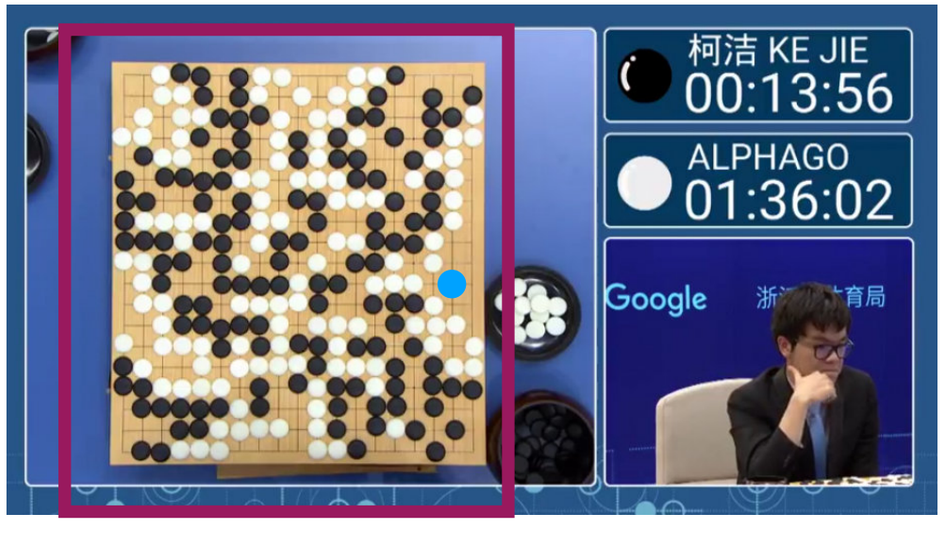


- Conduct action in any state of an environment.



In most problems, the agent needs to do a sequence of actions w.r.t. a sequence of states.

The Decision Problem

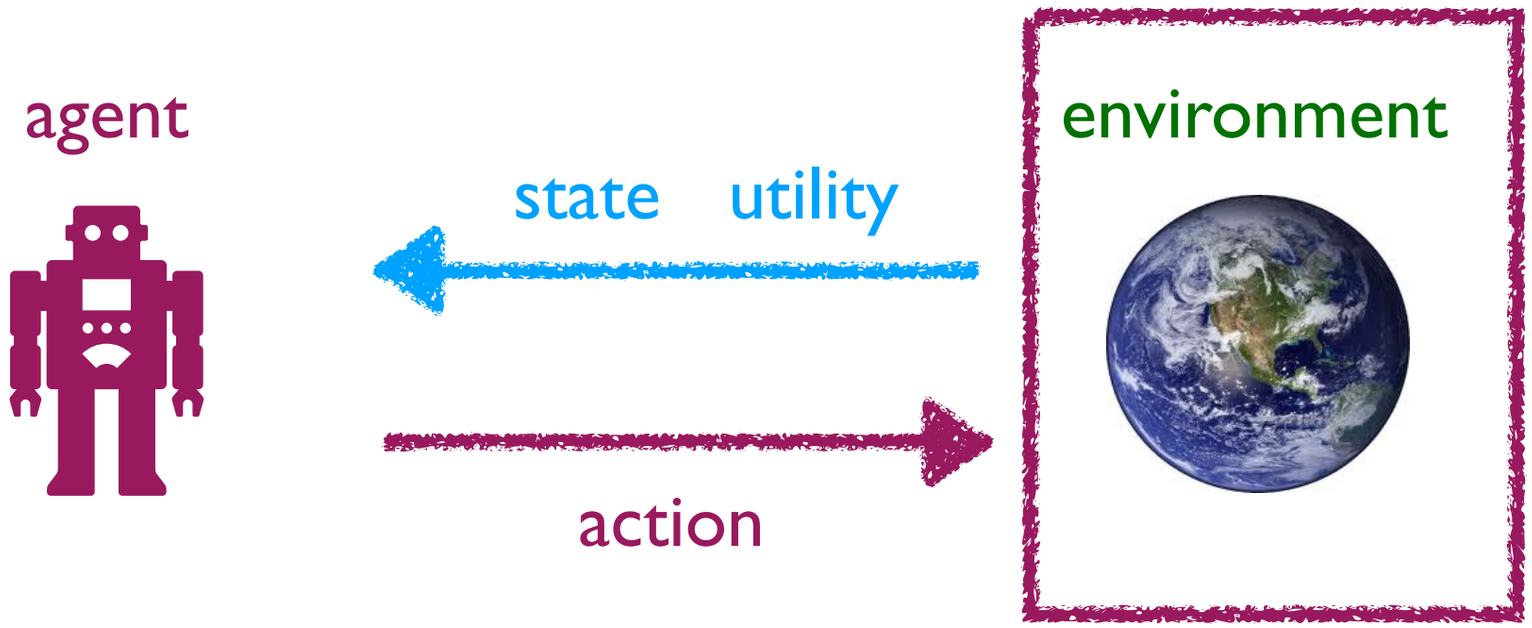


- The agent faces with a series of "states".
- Need to choose the corresponding "actions".
- Each action has a utility/cost.

we given utility a name: reward.

- Target: maximize the total reward in a decision sequence by always choosing the right action.

Model of the Environment



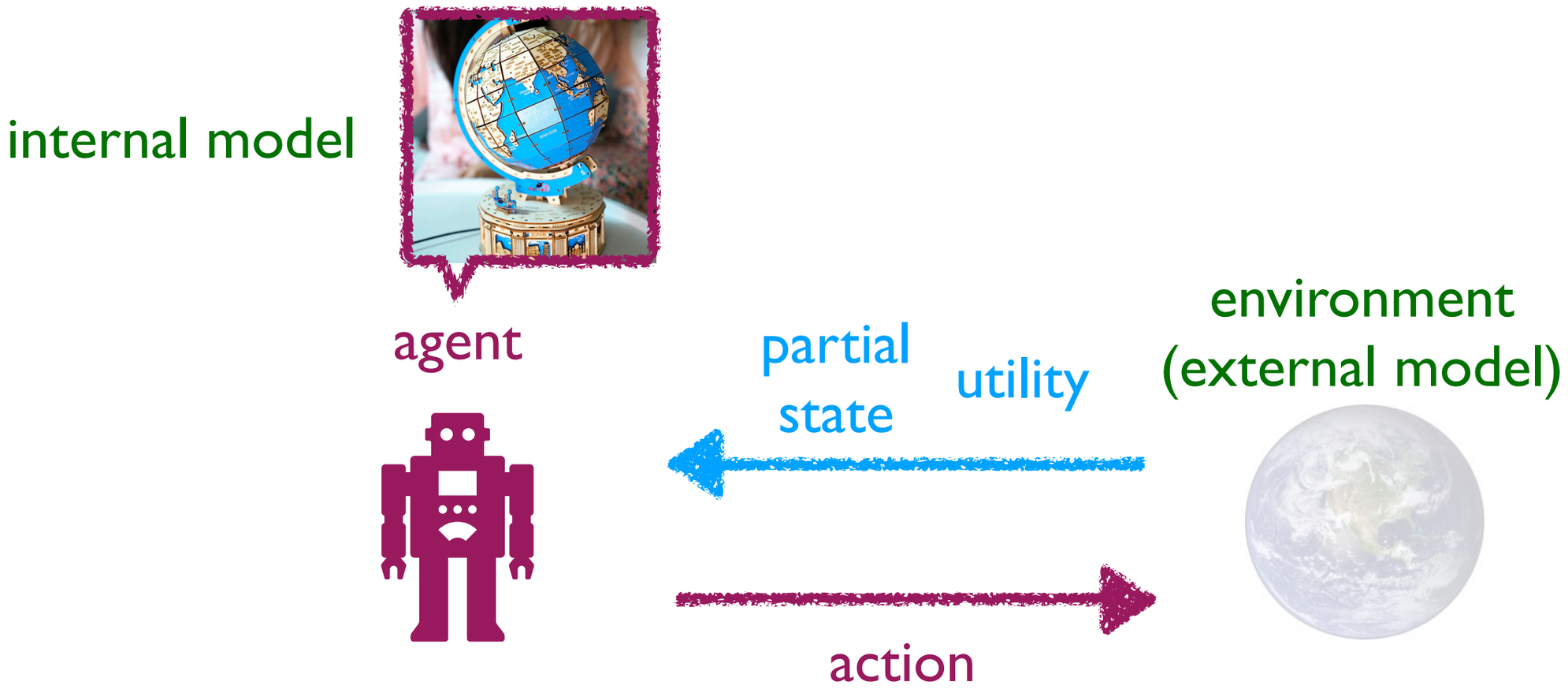
To make decisions in the environment, the agent usually needs a model of the environment to know how the things go on.
Where does this model come from?
Given by the problem (external) or built by the agent? (internal)

Internal vs. External Model

If the real environments known by the agent (e.g. WeiQi),
then the problem can be solved by dynamic programming.

If the real environment is unknown to the agent, it builds its internal model,
or even operates without model using reinforcement learning.

Reinforcement learning is approximated dynamic programming.

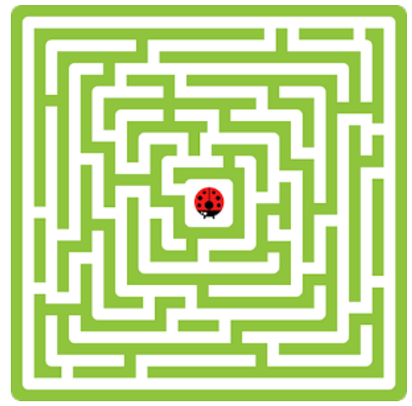


Reinforcement Learning

- Decision making is to find the optimal policy:
    - Decide best actions on all states.
    - No labeled <state, action> data, only receive reward.
        - The target is to maximize the long term total reward.
- Search-based decision making:
    - When the model is known, and the search cost is reasonable.
- Reinforcement learning:
    - Decision making in unknown model or search cost is too high.

Markov Decision Process

- How to model a maze problem?
- State: the current position.
- Action: left, right, up, down.



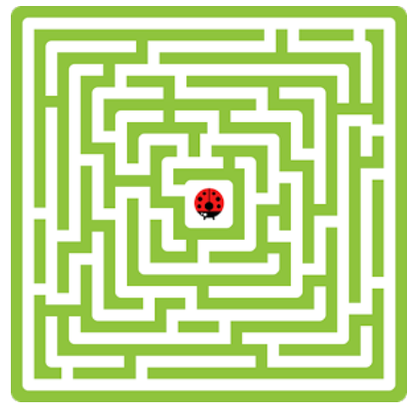
- Transition: where is the next position when take an action?
- Reward: how good is it instantly when take an action?
- Discount factor: How much the current action influences future?

Markov decision process (MDP) is the decision making model in RL with specific assumptions.

Mathematical Formulation

- A Markov Decision Process (MDP) is a five-tuple

$$< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$$



- $\mathcal{S}$ — The space of possible states (cont. or discrete)
- $\mathcal{A}$ — The space of possible actions (cont. or discrete)
- $\mathcal{P} : p(s_{t+1}|s_t, a_t)$ — The transition function (distribution)
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ — The reward function
- γ — The discount factor of rewards

The transition and reward functions can be stochastic!

The Markov Property

"The future is independent of the past given the present"

- The Markov property:

$$p(s_{t+1}|s_t, a_t) = p(s_{t+1}|s_1, s_2, ...s_t, a_t)$$

- The next state is only decided by the current state and action.
- The current state is a sufficient statistic.
- Non-Markovian decision problem:

小张出生于中国，2016年来到美国留学。小张的母语是(？)

The Learning Agent

- The agent takes a series of actions, experiences a series of states, and receives a series of rewards:

$$\{s_1, a_1, r_1\}, \{s_2, a_2, r_2\}, \{s_3, a_3, r_3\}...$$

- policy: function $p(a|s)$ used to select actions on any states.

- The target is to find the optimal policy to maximize the discounted total reward along the timeline:

$$r_1 + \gamma r_2 + \gamma^2 r_3 + ... = \sum_{t=1}^{\infty} \gamma^{t-1} r_t$$

- The discount factor measures how much the long term effect of the current action is concerned.

Value Functions: State Value Function V

- The state value function of a given policy is the expected total reward start from a given state, then follow the policy:

$$V_\pi(s) = \mathbb{E}_{\pi, p(s|s,a)}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right]$$

- The optimal policy have the optimal value function:

$$V_{\pi^*}(s) = \max_\pi V_\pi(s)$$

Value Functions: Action-State Value Function Q

- The action-state value function of a given policy is the expected total reward start from a given state, execute a given action, then follow the policy:

$$Q_\pi(s,a) = \mathbb{E}_{\pi, p(s|s,a)}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right]$$

- The optimal deterministic policy chooses the optimal action:

$$\pi^*(s) = \arg\max_a Q_{\pi^*}(s,a)$$

If the optimal action-state value function is known, so is the optimal policy!

Bellman Equation

- For the state value function,

$$V_\pi(s) = \mathbb{E}_{\pi(s),p(s|s,a)}\Big[\sum_{t=0}^{\infty}\gamma^t r_t|s_0 = s\Big]$$

$$= \mathbb{E}_{\pi(s_0),p(s_1|s_0,a_0)}\Big[r_0 + \gamma\mathbb{E}_{\pi(s),p(s|s,a)}\big[\sum_{t=1}^{\infty}\gamma^{t-1} r_t|s_1\big]|s_0 = s\Big]$$

$$= \mathbb{E}_{\pi(s_0),p(s_1|s_0,a_0)}\Big[r_0 + \gamma V_\pi(s_1)|s_0 = s\Big]$$

Recursive Definition

- For discrete state and action, and deterministic policy,

$$V_\pi(s) = \sum_{s'} p\big(s'|s,\pi(s)\big)\Big[r\big(s,\pi(s),s'\big) + \gamma V_\pi(s')\Big]$$

Bellman Equation (Cont.)

- For the action-state value function,

$$Q_\pi(s,a) = \mathbb{E}_{\pi(s),p(s|s,a)}\Big[\sum_{t=0}^{\infty}\gamma^t r_t|s_0=s,a_0=a\Big]$$

$$= \mathbb{E}_{\pi(s_0),p(s_1|s_0,a_0)}\Big[r_0+\gamma\mathbb{E}_{\pi(s),p(s|s,a)}\big[\sum_{t=1}^{\infty}\gamma^{t-1}r_t|s_1,a_1\big]|s_0=s,a_0=a\Big]$$

$$= \mathbb{E}_{\pi(s_0),p(s_1|s_0,a_0)}\Big[r_0+\gamma Q_\pi(s_1,a_1)|s_0=s,a_0=a\Big]$$

Recursive Definition

- For discrete state and action, and deterministic policy,

$$Q_\pi(s,a) = \sum_{s'} p\big(s'|s,a\big)\Big[r(s,a,s')+\gamma Q_\pi\big(s',\pi(s')\big)\Big]$$

Bellman Equation (Cont.)

- For optimal deterministic policy π^*,

$$V_{\pi^*}(s) = \max_a \Big[\sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V_{\pi^*}(s')] \Big]$$

$$Q_{\pi^*}(s,a) = \sum_{s'} p(s'|s,a) \Big[r(s,a,s') + \gamma \max_{a'} Q_{\pi^*}(s',a') \Big]$$

- Then the optimal deterministic policy is

$$\pi^*(s) = \arg\max_a Q_{\pi^*}(s,a)$$

- Due the recursive structure, the optimal value functions can be solved by dynamical programming. This assumes that the full information of the MDP is known!

Basic Assumptions

- For simplicity, assume discrete state and action, deterministic transition and policy. Then

$$V^*(s) = \arg\max_a Q^*(s,a)$$

$$\begin{aligned} Q^*(s,a) &= r(s,a) + \gamma[\arg\max_{a'} Q^*(s',a')] \\ &= r(s,a) + \gamma[V^*(s')] \end{aligned}$$

- The two value functions can be computed from each other.
- $Q_\pi(s,a)$ is easier for choosing actions, while the space is $\mathcal{S} \times \mathcal{A}$
- $V_\pi(s)$ has much smaller space.

Solving MDP:Value Iteration

- Initialize value function V_0
- For i=1,2,3… until convergence
    - Update V_i for each state

Why iterative update?
Loop exists in the MDP!

$$V_i(s) = \max_a \left[\sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V_{i-1}(s')] \right]$$

- Theoretical convergence guarantee to V^* and π^*

Policy Iteration

- For discrete state and action, deterministic transition and deterministic policy:

$$V^*(s) = \arg\max_a Q^*(s,a)$$

$$\begin{aligned} Q^*(s,a) &= r(s,a) + \gamma[\arg\max_{a'} Q^*(s',a')] \\ &= r(s,a) + \gamma[V^*(s')] \end{aligned}$$

- The two value functions can be computed from each other.
- $Q_\pi(s,a)$ is easier for choosing actions, while the space is $\mathcal{S} \times \mathcal{A}$
- $V_\pi(s)$ has much smaller space.

Solving MDP: Policy Iteration

- Initialize value function V_0 and policy π_0
- For i=1,2,3… until convergence
  - Policy evaluation step: update V_i for each state

$$V_i(s) = r + \gamma V_{i-1}^{\pi_{i-1}}(s')$$

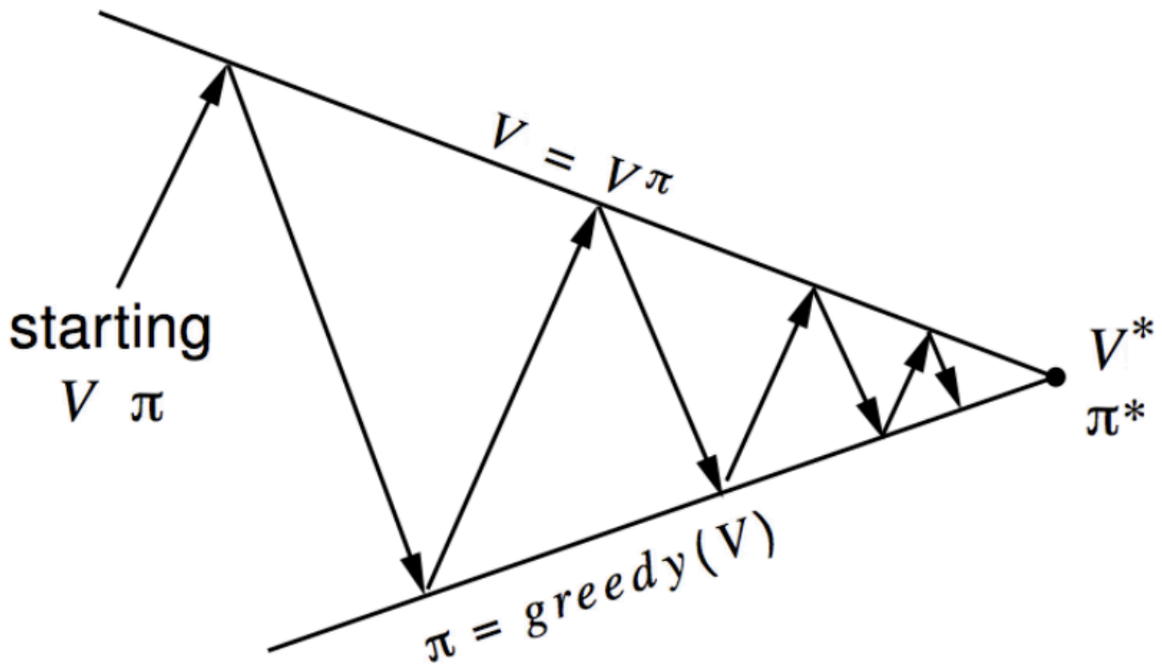
Calculated based on π_{i-1}
Actually an inner loop to do iterative update until convergence

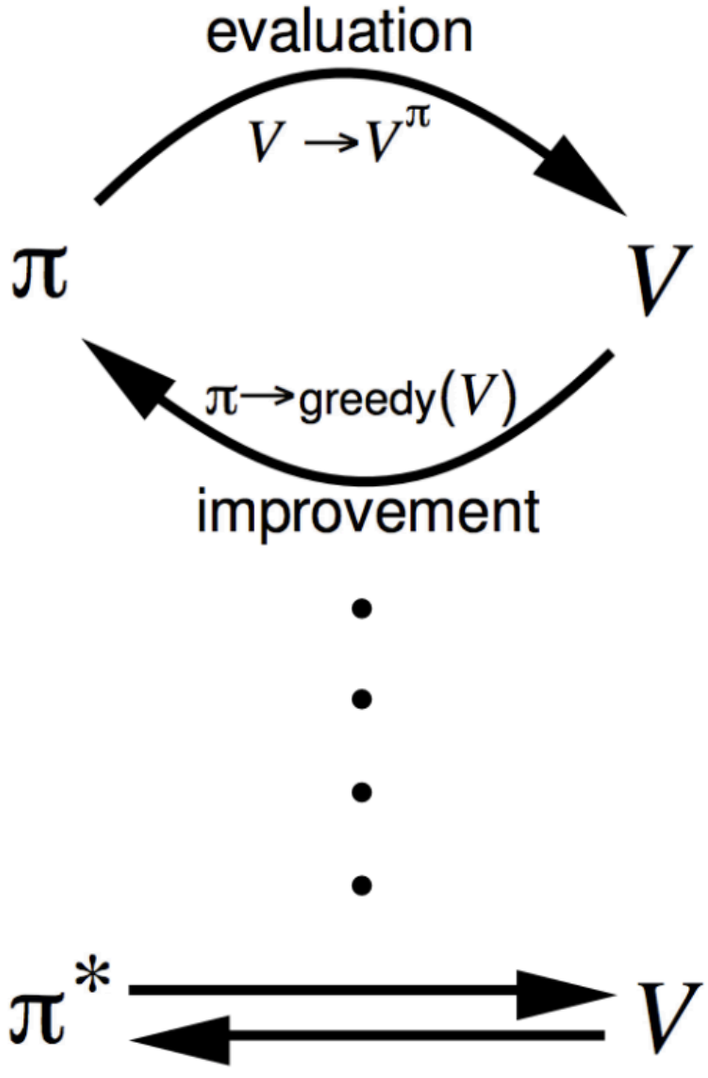
  - Policy improvement step: update π_i for each s-a pair.

$$\pi(s) \leftarrow arg\max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$$

- Theoretical convergence guarantee to V^* and π^*

Policy Iteration (Cont.)





Policy evaluation Estimate v_π
Iterative policy evaluation

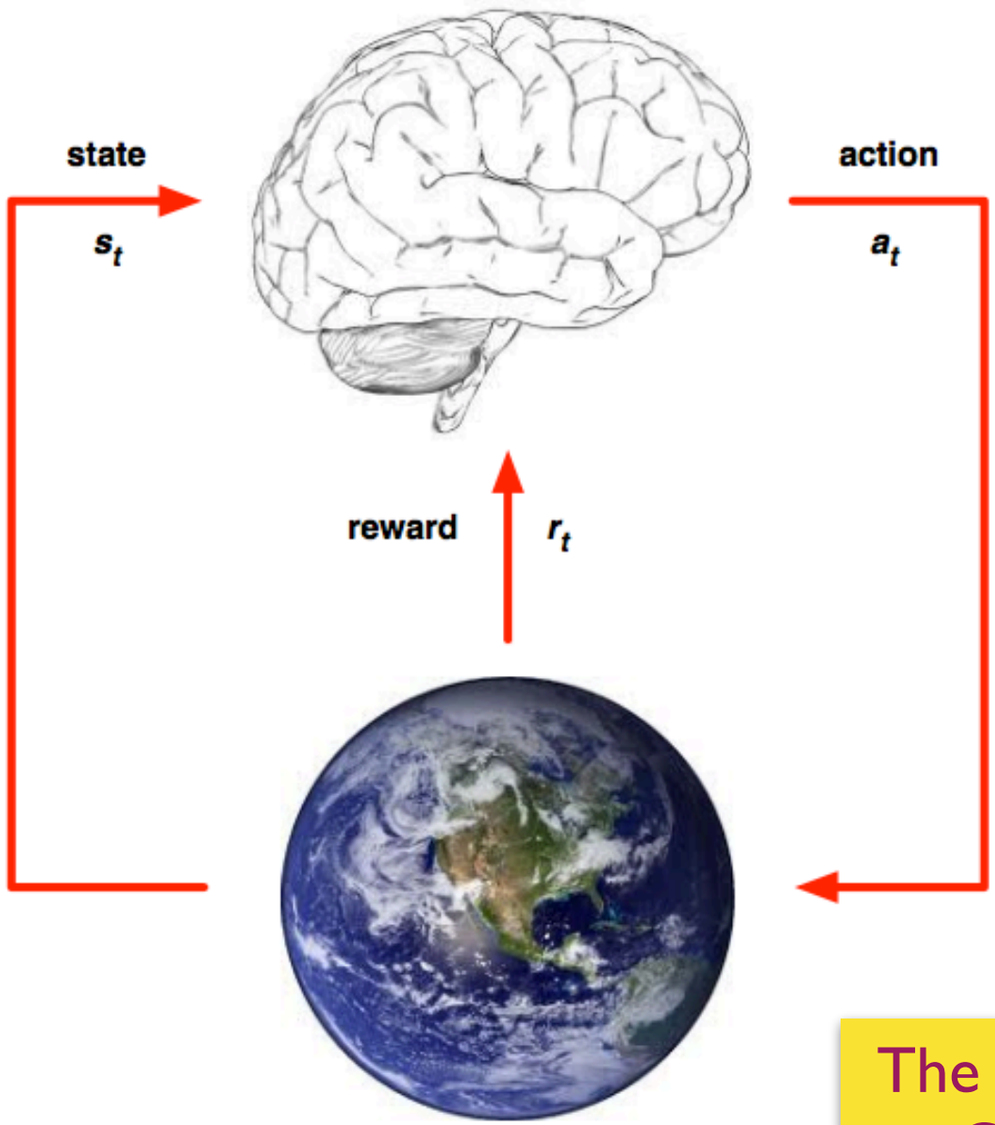
Policy improvement Generate $\pi' \geq \pi$
Greedy policy improvement

Slide courtesy: David Silver

Reinforcement Learning

- When full information of the MDP is known, the value function can be solved by planning.
- But how to solve when not fully known? $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$
- In RL, usually the state transition $\mathcal{P}$ and reward function $\mathcal{R}$ are not known.
- The agent has to learn by trial and error, facing with the exploration and exploitation problem.

Agent and Environment



- At each step t the agent:
  - Receives state s_t
  - Receives scalar reward r_t
  - Executes action a_t
- The environment:
  - Receives action a_t
  - Emits state s_t
  - Emits scalar reward r_t

The agent can only interact with true environment.
Can not use model for searching or planning.

- The target is still to learn the optimal value function.

Slide courtesy: David Silver

Basic Idea

- Value function based RL aims at estimating the optimal value function.

- Value function update: update using new estimation

$$Q_{i+1}(s,a) = (1-\alpha)Q_i(s,a) + \alpha\hat{Q}_i(s,a)$$

  - Monte-Carlo RL — Estimate by sampled trajectories

  - Temporal difference Learning — SARSA and Q-learning.

- Policy Improvement:

  - Based on new value function, with ϵ - greedy.

Monte-Carlo RL

- Given policy π_i, we can sample trajectories:

$$\{s_1, a_1, r_1\}, \{s_2, a_2, r_2\}, \{s_3, a_3, r_3\}...$$

- Then we can get empirical estimate:

$$\hat{Q}_i(s_1, a_1) = r_1 + \gamma r_2 + \gamma^2 r_3 + ...$$

Can we still update the policy in greedy?

No!

- Update value function:

$$Q_{i+1}(s, a) = (1 - \alpha)Q_i(s, a) + \alpha\hat{Q}_i(s, a)$$

- Follow the spirit of policy iteration, update $\pi_i \rightarrow \pi_{i+1}$

Exploration vs. Exploitation

"Behind one door is tenure - behind the other is flipping burgers at McDonald's."

- There are two doors in front of you.
- You open the left door and get reward 0
  $V(left) = 0$
- You open the right door and get reward $+1$
  $V(right) = +1$
- You open the right door and get reward $+3$
  $V(right) = +2$
- You open the right door and get reward $+2$
  $V(right) = +2$

$\vdots$

- Are you sure you've chosen the best door?

Slide courtesy by David Silver

Exploration vs. Exploitation (Cont.)

- In policy iteration, the policy improvement step is greedy:

$$\pi_i(s) = \arg\max_a Q_i(s, a)$$

- But for RL, since the environment is not fully known, greedy update may perform arbitrarily bad — need to allow some exploration.

- Common choice: use ϵ- greedy policy:

with prob. $1 - \epsilon$, execute as greedy

with prob. ϵ, execute randomly

- Theoretical guarantee: If the exploration vanishes, we can ensure convergence.

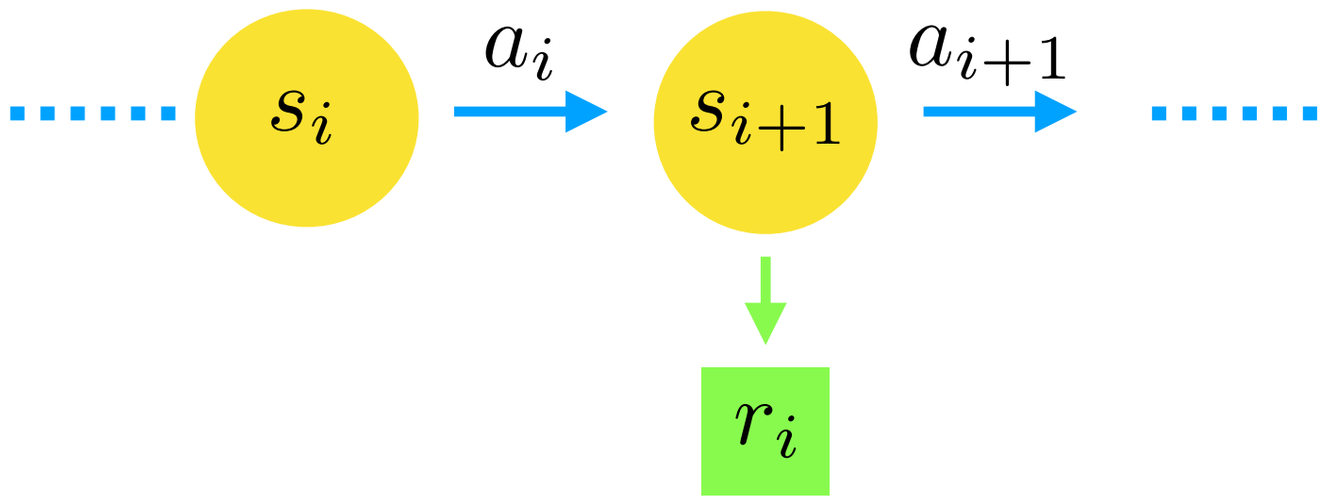
TD vs. MC

- Temporal-difference (TD) learning has several advantages over Monte-Carlo (MC)
  - Lower variance
  - Online
  - Incomplete sequences
- Natural idea: use TD instead of MC in our control loop
  - Apply TD to $Q(S, A)$
  - Use ϵ-greedy policy improvement
  - Update every time-step

Slide courtesy: David Silver

SARSA

- "State-Action-Reward-State-Action" — SARSA



Run the current best ϵ-greedy policy

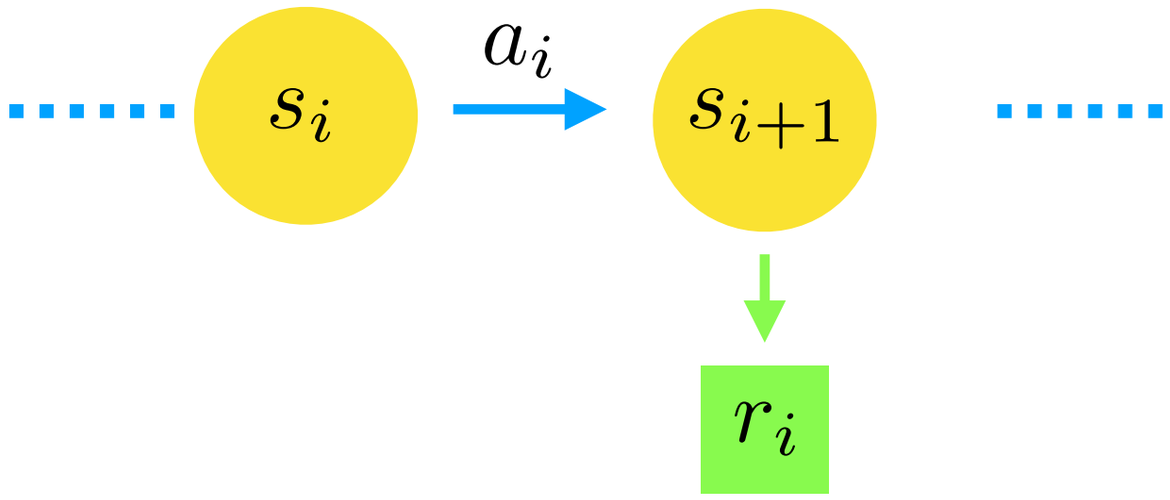
- Once collect s-a-r-s-a sample, do value function update:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \Big[r_i + \gamma Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i) \Big]$$

TD error

Always use the policy on-the-run, called "on-policy"

Q-Learning



Run the current
best ϵ-greedy policy

- Once collect s-a-r-s sample, do value function update:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha\Big[r_i + \gamma Q(s_{i+1}, \hat{\pi}^*(s_{i+1})) - Q(s_i, a_i)\Big]$$

The current best policy

The policy on the run can be different from
the current best policy in the update, called "off-policy"

Off-Policy Learning

- Evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s, a)$
- While following behaviour policy $\mu(a|s)$

$$\{S_1, A_1, R_2, ..., S_T\} \sim \mu$$

- Why is this important?
- Learn from observing humans or other agents
- Re-use experience generated from old policies $\pi_1, \pi_2, ..., \pi_{t-1}$
- Learn about *optimal* policy while following *exploratory* policy
- Learn about *multiple* policies while following *one* policy

Slide courtesy: David Silver

Those who cannot remember the past are condemned to repeat it.

- Dynamic Programming

Try to do more exercises.

Thanks for your attention!
Discussions?

Reference

Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani: Chap. 6.

Algorithm design: Chap. 6.

https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/06DynamicProgrammingI.pdf

https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/06DynamicProgrammingII.pdf

Introduction to algorithms (4th edition): Chap. 14.

Data Structure and Algorithm Analysis in C (2nd Edition)： Sec. 10.3.

https://www.davidsilver.uk/teaching/ Lec. 1-5 (about reinforcement learning)